Learn to Think Like a Programmer

JAVA AND ALGORITHMIC THINKING FOR THE COMPLETE BEGINNER

3rd REVISED EDITION

ARISTIDES S. BOURAS

FOR LINUX & WINDOWS USERS

For optimal reading experience, please rotate your device to landscape orientation.

Table of Contents

- Table of Contents
- Preface
    - About the Author
    - Acknowledgments
    - How This Book is Organized
    - Who Should Buy This Book?
    - Conventions Used in This Book
    - How to Report Errata
    - Where to Download Material About this Book
    - If you Like this Book
- Part I Introductory Knowledge
    - Chapter 1 How a Computer Works
        - 1.1 Introduction
        - 1.2 What is Hardware?
        - 1.3 What is Software?
        - 1.4 How a Computer Executes (Runs) a Program
        - 1.5 Compilers and Interpreters
        - 1.6 What is Source Code?
        - 1.7 Review Questions: True/False
        - 1.8 Review Questions: Multiple Choice
    - Chapter 2 Java and Integrated Development Environments
        - 2.1 What is Java?
        - 2.2 What is the Difference Between a Script and a Program?
        - 2.3 Why You Should Learn Java
        - 2.4 How Java Works
        - 2.5 Java Development Kit (JDK)
        - 2.6 Integrated Development Environments

        - 2.7 Microsoft Visual Studio
    - Chapter 3 Software Packages to Install
        - 3.1 What to Install
    - Review in “Introductory Knowledge”
        - Review Crossword Puzzles
        - Review Questions
- Part II Getting Started with Java
    - Chapter 4 Introduction to Basic Algorithmic Concepts
        - 4.1 What is an Algorithm?
        - 4.2 The Algorithm for Making a Cup of Tea
        - 4.3 Properties of an Algorithm
        - 4.4 Okay About Algorithms. But What is a Computer Program Anyway?
        - 4.5 The Three Parties!
        - 4.6 The Three Main Stages Involved in Creating an Algorithm
        - 4.7 Flowcharts
            - Exercise 4.7-1 Finding the Average Value of Three Numbers
        - 4.8 What are ”Reserved Words”?
        - 4.9 What is the Difference Between a Statement and a Command?
        - 4.10 What is Structured Programming?
        - 4.11 The Three Fundamental Control Structures
            - Exercise 4.11-1 Understanding Control Structures Using Flowcharts
        - 4.12 Your First Java Program

- 4.13 What is the Difference Between a Syntax Error, a Logic Error, and a Runtime Error?
- 4.14 What “Debugging” Means
- 4.15 Commenting Your Code
- 4.16 User-Friendly Programs
- 4.17 Review Questions: True/False
- 4.18 Review Questions: Multiple Choice

- Chapter 5 Variables and Constants

  - 5.1 What is a Variable?
  - 5.2 What is a Constant?
  - 5.3 How Many Types of Variables and Constants Exist?
  - 5.4 Rules and Conventions for Naming Variables and Constants in Java
  - 5.5 What Does the Phrase “Declare a Variable” Mean?
  - 5.6 How to Declare Variables in Java
  - 5.7 How to Declare Constants in Java
  - 5.8 Review Questions: True/False
  - 5.9 Review Questions: Multiple Choice
  - 5.10 Review Exercises

- Chapter 6 Handling Input and Output

  - 6.1 How to Output Messages and Results to a User's Screen?
  - 6.2 How to Output Special Characters?
  - 6.3 How to Prompt the User to Enter Data?
  - 6.4 Review Questions: True/False
  - 6.5 Review Questions: Multiple Choice

- Chapter 7 Operators

  - 7.1 The Value Assignment Operator
  - 7.2 Arithmetic Operators
  - 7.3 What is the Precedence of Arithmetic Operators?
  - 7.4 Compound Assignment Operators

        - Exercise 7.4-1 Which Java Statements are Syntactically Correct?
        - Exercise 7.4-2 Finding Variable Types
    - 7.5 Incrementing/Decrementing Operators
    - 7.6 String Operators
        - Exercise 7.6-1 Concatenating Names
    - 7.7 Review Questions: True/False
    - 7.8 Review Questions: Multiple Choice
    - 7.9 Review Exercises
  - Chapter 8 Trace Tables
    - 8.1 What is a Trace Table?
        - Exercise 8.1-1 Creating a Trace Table
        - Exercise 8.1-2 Creating a Trace Table
        - Exercise 8.1-3 Swapping Values of Variables
        - Exercise 8.1-4 Swapping Values of Variables – An Alternative Approach
    - 8.2 Review Questions: True/False
    - 8.3 Review Exercises
  - Chapter 9 Using Visual Studio Code
    - 9.1 Write, Execute and Debug Java Programs
  - Review in “Getting Started with Java”
    - Review Crossword Puzzles
    - Review Questions
- Part III Sequence Control Structures
  - Chapter 10 Introduction to Sequence Control Structures

- 10.1 What is the Sequence Control Structure?
  - Exercise 10.1-1 Calculating the Area of a Rectangle
  - Exercise 10.1-2 Calculating the Area of a Circle
  - Exercise 10.1-3 Where is the Car? Calculating Distance Traveled
  - Exercise 10.1-4 Kelvin to Fahrenheit
  - Exercise 10.1-5 Calculating Sales Tax
  - Exercise 10.1-6 Calculating a Sales Discount
  - Exercise 10.1-7 Calculating a Sales Discount and Tax
- 10.2 Review Exercises

- Chapter 11 Manipulating Numbers
  - 11.1 Introduction
  - 11.2 Useful Mathematical Methods (Subprograms), and More
    - Exercise 11.2-1 Calculating the Distance Between Two Points
    - Exercise 11.2-2 How Far Did the Car Travel?
  - 11.3 Review Questions: True/False
  - 11.4 Review Questions: Multiple Choice
  - 11.5 Review Exercises

- Chapter 12 Complex Mathematical Expressions
  - 12.1 Writing Complex Mathematical Expressions
    - Exercise 12.1-1 Representing Mathematical Expressions in Java
    - Exercise 12.1-2 Writing a Mathematical Expression in Java
    - Exercise 12.1-3 Writing a Complex Mathematical Expression in Java

        - 12.2 Review Exercises

    - Chapter 13 Exercises With a Quotient and a Remainder

        - 13.1 Introduction

            - Exercise 13.1-1 Calculating the Quotient and Remainder of Integer Division
            - Exercise 13.1-2 Finding the Sum of Digits
            - Exercise 13.1-3 Displaying an Elapsed Time
            - Exercise 13.1-4 Reversing a Number

        - 13.2 Review Exercises

    - Chapter 14 Manipulating Strings

        - 14.1 Introduction
        - 14.2 The Position of a Character in a String
        - 14.3 Useful String Methods (Subprograms), and More

            - Exercise 14.3-1 Displaying a String Backwards
            - Exercise 14.3-2 Switching the Order of Names
            - Exercise 14.3-3 Creating a Login ID
            - Exercise 14.3-4 Creating a Random Word
            - Exercise 14.3-5 Finding the Sum of Digits

        - 14.4 Review Questions: True/False
        - 14.5 Review Questions: Multiple Choice
        - 14.6 Review Exercises

    - Review in “Sequence Control Structures”

        - Review Crossword Puzzle
        - Review Questions

- Part IV Decision Control Structures

    - Chapter 15 Making Questions

- 15.1 Introduction
- 15.2 What is a Boolean Expression?
- 15.3 How to Write Simple Boolean Expressions
    - Exercise 15.3-1 Filling in the Table
- 15.4 Logical Operators and Complex Boolean Expressions
    - Exercise 15.4-1 Calculating the Results of Complex Boolean Expressions
- 15.5 Assigning the Result of a Boolean Expression to a Variable
- 15.6 What is the Order of Precedence of Logical Operators?
    - Exercise 15.6-1 Filling in the Truth Table
    - Exercise 15.6-2 Converting English Sentences to Boolean Expressions
- 15.7 What is the Order of Precedence of Arithmetic, Comparison, and Logical Operators?
- 15.8 How to Negate Boolean Expressions
    - Exercise 15.8-1 Negating Boolean Expressions
- 15.9 Review Questions: True/False
- 15.10 Review Questions: Multiple Choice
- 15.11 Review Exercises

- Chapter 16 The Single-Alternative Decision Structure
    - 16.1 The Single-Alternative Decision Structure
        - Exercise 16.1-1 Trace Tables and Single-Alternative Decision Structures
        - Exercise 16.1-2 The Absolute Value of a Number
    - 16.2 Review Questions: True/False
    - 16.3 Review Questions: Multiple Choice

- 16.4 Review Exercises

- Chapter 17 The Dual-Alternative Decision Structure
  - 17.1 The Dual-Alternative Decision Structure
    - Exercise 17.1-1 Finding the Output Message
    - Exercise 17.1-2 Trace Tables and Dual-Alternative Decision Structures
    - Exercise 17.1-3 Who is the Greatest?
    - Exercise 17.1-4 Finding Odd and Even Numbers
    - Exercise 17.1-5 Weekly Wages
  - 17.2 Review Questions: True/False
  - 17.3 Review Questions: Multiple Choice
  - 17.4 Review Exercises

- Chapter 18 The Multiple-Alternative Decision Structure
  - 18.1 The Multiple-Alternative Decision Structure
    - Exercise 18.1-1 Trace Tables and Multiple-Alternative Decision Structures
    - Exercise 18.1-2 Counting the Digits
  - 18.2 Review Questions: True/False
  - 18.3 Review Exercises

- Chapter 19 The Case Decision Structure
  - 19.1 The Case Decision Structure
    - Exercise 19.1-1 The Days of the Week
  - 19.2 Review Questions: True/False
  - 19.3 Review Exercises

- Chapter 20 Nested Decision Control Structures

        - 20.1 What are Nested Decision Control Structures?
            - Exercise 20.1-1 Trace Tables and Nested Decision Control Structures
            - Exercise 20.1-2 Positive, Negative or Zero?
        - 20.2 Review Questions: True/False
        - 20.3 Review Exercises
    - Chapter 21 More about Flowcharts with Decision Control Structures
        - 21.1 Introduction
        - 21.2 Converting Java Programs to Flowcharts
            - Exercise 21.2-1 Designing the Flowchart
            - Exercise 21.2-2 Designing the Flowchart
            - Exercise 21.2-3 Designing the Flowchart
        - 21.3 A Mistake That You Will Probably Make!
        - 21.4 Converting Flowcharts to Java Programs
            - Exercise 21.4-1 Writing the Java Program
            - Exercise 21.4-2 Writing the Java Program
            - Exercise 21.4-3 Writing the Java Program
        - 21.5 Review Exercises
    - Chapter 22 Tips and Tricks with Decision Control Structures
        - 22.1 Introduction
        - 22.2 Choosing a Decision Control Structure
        - 22.3 Streamlining the Decision Control Structure
            - Exercise 22.3-1 “Shrinking” the Algorithm
            - Exercise 22.3-2 “Shrinking” the Java Program
            - Exercise 22.3-3 “Shrinking” the Algorithm

- 22.4 Logical Operators – to Use, or not to Use: That is the Question!
  - Exercise 22.4-1 Rewriting the Code
  - Exercise 22.4-2 Rewriting the Code
- 22.5 Merging Two or More Single-Alternative Decision Structures
  - Exercise 22.5-1 Merging the Decision Control Structures
  - Exercise 22.5-2 Merging the Decision Control Structures
- 22.6 Replacing Two Single-Alternative Decision Structures with a Dual-Alternative One
  - Exercise 22.6-1 “Merging” the Decision Control Structures
- 22.7 Put the Boolean Expressions Most Likely to be True First
  - Exercise 22.7-1 Rearranging the Boolean Expressions
- 22.8 Why is Code Indentation so Important?
- 22.9 Review Questions: True/False
- 22.10 Review Questions: Multiple Choice
- 22.11 Review Exercises

Chapter 23 More with Decision Control Structures

- 23.1 Simple Exercises with Decision Control Structures
  - Exercise 23.1-1 Is it an Integer?
  - Exercise 23.1-2 Validating Data Input and Finding Odd and Even Numbers
  - Exercise 23.1-3 Where is the Tollkeeper?
  - Exercise 23.1-4 The Most Scientific Calculator Ever!

    - Exercise 23.1-5 Converting Gallons to Liters, and Vice Versa
    - Exercise 23.1-6 Converting Gallons to Liters, and Vice Versa (with Data Validation)

- 23.2 Finding Minimum and Maximum Values with Decision Control Structures

    - Exercise 23.2-1 Finding the Name of the Heaviest Person

- 23.3 Decision Control Structures in Solving Mathematical Problems

    - Exercise 23.3-1 Finding the Value of y
    - Exercise 23.3-2 Finding the Values of y
    - Exercise 23.3-3 Solving the Linear Equation ax + b = 0
    - Exercise 23.3-4 Solving the Quadratic Equation ax2 + bx + c = 0

- 23.4 Exercises with Series of Consecutive Ranges of Values

    - Exercise 23.4-1 Calculating the Discount
    - Exercise 23.4-2 Validating Data Input and Calculating the Discount
    - Exercise 23.4-3 Sending a Parcel
    - Exercise 23.4-4 Finding the Values of y
    - Exercise 23.4-5 Progressive Rates and Electricity Consumption
    - Exercise 23.4-6 Progressive Rates and Text Messaging Services

- 23.5 Exercises of a General Nature with Decision Control Structures

    - Exercise 23.5-1 Finding a Leap Year
    - Exercise 23.5-2 Displaying the Days of the Month

            - Exercise 23.5-3 Checking for Proper Capitalization and Punctuation
            - Exercise 23.5-4 Is the Number a Palindrome?

        - 23.6 Boolean Expressions Reference and Handy Tips
        - 23.7 Review Exercises

    - Review in “Decision Control Structures”

        - Review Crossword Puzzle
        - Review Questions

- Part V Loop Control Structures

    - Chapter 24 Introduction to Loop Control Structures

        - 24.1 What is a Loop Control Structure?
        - 24.2 From Sequence Control to Loop Control Structures
        - 24.3 Review Questions: True/False

    - Chapter 25 Pre-Test, Mid-Test and Post-Test Loop Structures

        - 25.1 The Pre-Test Loop Structure

            - Exercise 25.1-1 Designing the Flowchart and Counting the Total Number of Iterations
            - Exercise 25.1-2 Counting the Total Number of Iterations
            - Exercise 25.1-3 Counting the Total Number of Iterations
            - Exercise 25.1-4 Counting the Total Number of Iterations
            - Exercise 25.1-5 Finding the Sum of Four Numbers
            - Exercise 25.1-6 Finding the Sum of Odd Numbers
            - Exercise 25.1-7 Finding the Sum of N Numbers
            - Exercise 25.1-8 Finding the Sum of an Unknown Quantity of Numbers
            - Exercise 25.1-9 Finding the Product of 20 Numbers

- 25.2 The Post-Test Loop Structure
  - Exercise 25.2-1 Designing the Flowchart and Counting the Total Number of Iterations
  - Exercise 25.2-2 Counting the Total Number of Iterations
  - Exercise 25.2-3 Designing the Flowchart and Counting the Total Number of Iterations
  - Exercise 25.2-4 Counting the Total Number of Iterations
  - Exercise 25.2-5 Finding the Product of N Numbers
- 25.3 The Mid-Test Loop Structure
  - Exercise 25.3-1 Designing the Flowchart and Counting the Total Number of Iterations
- 25.4 Review Questions: True/False
- 25.5 Review Questions: Multiple Choice
- 25.6 Review Exercises

◦ Chapter 26 Definite Loops

- 26.1 The for statement
  - Exercise 26.1-1 Creating the Trace Table
  - Exercise 26.1-2 Creating the Trace Table
  - Exercise 26.1-3 Counting the Total Number of Iterations
  - Exercise 26.1-4 Finding the Sum of Four Numbers
  - Exercise 26.1-5 Finding the Square Roots from 0 to N
  - Exercise 26.1-6 Finding the Sum of 1 + 2 + 3 + … + 100
  - Exercise 26.1-7 Finding the Product of 2 × 4 × 6 × 8 × 10
  - Exercise 26.1-8 Finding the Sum of 22 + 42 + 62 + … (2N)2

        - Exercise 26.1-9 Finding the Sum of 33 + 66 + 99 + … (3N)3N
        - Exercise 26.1-10 Finding the Average Value of Positive Numbers
        - Exercise 26.1-11 Counting the Vowels

    - 26.2 Rules that Apply to For-Loops

        - Exercise 26.2-1 Counting the Total Number of Iterations
        - Exercise 26.2-2 Counting the Total Number of Iterations
        - Exercise 26.2-3 Counting the Total Number of Iterations
        - Exercise 26.2-4 Counting the Total Number of Iterations
        - Exercise 26.2-5 Finding the Sum of N Numbers

    - 26.3 Review Questions: True/False
    - 26.4 Review Questions: Multiple Choice
    - 26.5 Review Exercises

- Chapter 27 Nested Loop Control Structures

    - 27.1 What is a Nested Loop?

        - Exercise 27.1-1 Say “Hello Zeus”. Counting the Total Number of Iterations.
        - Exercise 27.1-2 Creating the Trace Table

    - 27.2 Rules that Apply to Nested Loops

        - Exercise 27.2-1 Violating the First Rule
        - Exercise 27.2-2 Violating the Second Rule

    - 27.3 Review Questions: True/False
    - 27.4 Review Questions: Multiple Choice
    - 27.5 Review Exercises

- Chapter 28 More about Flowcharts with Loop Control Structures
  - 28.1 Introduction
  - 28.2 Converting Java Programs to Flowcharts
    - Exercise 28.2-1 Designing the Flowchart Fragment
    - Exercise 28.2-2 Designing the Flowchart Fragment
    - Exercise 28.2-3 Designing the Flowchart
    - Exercise 28.2-4 Designing the Flowchart Fragment
    - Exercise 28.2-5 Designing the Flowchart
  - 28.3 Converting Flowcharts to Java Programs
    - Exercise 28.3-1 Writing the Java Program
    - Exercise 28.3-2 Writing the Java Program
    - Exercise 28.3-3 Writing the Java Program
    - Exercise 28.3-4 Writing the Java Program
  - 28.4 Review Exercises
- Chapter 29 Tips and Tricks with Loop Control Structures
  - 29.1 Introduction
  - 29.2 Choosing a Loop Control Structure
  - 29.3 The “Ultimate” Rule
  - 29.4 Breaking Out of a Loop
  - 29.5 Cleaning Out Your Loops
    - Exercise 29.5-1 Cleaning Out the Loop
    - Exercise 29.5-2 Cleaning Out the Loop
  - 29.6 Endless Loops and How to Stop Them
  - 29.7 The “From Inner to Outer” Method
  - 29.8 Review Questions: True/False
  - 29.9 Review Questions: Multiple Choice
  - 29.10 Review Exercises
- Chapter 30 More with Loop Control Structures

- 30.1 Simple Exercises with Loop Control Structures
  - Exercise 30.1-1 Counting the Numbers According to Which is Greater
  - Exercise 30.1-2 Counting the Numbers According to Their Digits
  - Exercise 30.1-3 How Many Numbers Fit in a Sum
  - Exercise 30.1-4 Finding the Total Number of Positive Integers
  - Exercise 30.1-5 Iterating as Many Times as the User Wishes
  - Exercise 30.1-6 Finding the Sum of the Digits
- 30.2 Exercises with Nested Loop Control Structures
  - Exercise 30.2-1 Displaying all Three-Digit Integers that Contain a Given Digit
  - Exercise 30.2-2 Displaying all Instances of a Specified Condition
- 30.3 Data Validation with Loop Control Structures
  - Exercise 30.3-1 Finding Odd and Even Numbers
  - Exercise 30.3-2 Finding the Sum of Four Numbers
- 30.4 Finding Minimum and Maximum Values with Loop Control Structures
  - Exercise 30.4-1 Validating and Finding the Minimum and the Maximum Value
  - Exercise 30.4-2 Validating and Finding the Hottest Planet
  - Exercise 30.4-3 ”Making the Grade”
- 30.5 Using Loop Control Structures to Solve Mathematical Problems

        - Exercise 30.5-1 Calculating the Area of as Many Triangles as the User Wishes
        - Exercise 30.5-2 Finding x and y
        - Exercise 30.5-3 The Russian Multiplication Algorithm
        - Exercise 30.5-4 Finding the Number of Divisors
        - Exercise 30.5-5 Is the Number a Prime?
        - Exercise 30.5-6 Finding all Prime Numbers from 1 to N
        - Exercise 30.5-7 Heron's Square Root
        - Exercise 30.5-8 Calculating π
        - Exercise 30.5-9 Approximating a Real with a Fraction
    - 30.6 Exercises of a General Nature with Loop Control Structures
        - Exercise 30.6-1 Fahrenheit to Kelvin, from 0 to 100
        - Exercise 30.6-2 Rice on a Chessboard
        - Exercise 30.6-3 Just a Poll
        - Exercise 30.6-4 Is the Message a Palindrome?
    - 30.7 Review Questions: True/False
    - 30.8 Review Exercises
  - Review in “Loop Control Structures”
    - Review Crossword Puzzle
    - Review Questions
- Part VI Data Structures in Java
  - Chapter 31 One-Dimensional Arrays and HashMaps
    - 31.1 Introduction
    - 31.2 What is an Array?
        - Exercise 31.2-1 Designing an Array
        - Exercise 31.2-2 Designing Arrays

    - Exercise 31.2-3 Designing Arrays

- 31.3 Creating One-Dimensional Arrays in Java
- 31.4 How to Get Values from a One-Dimensional Array

    - Exercise 31.4-1 Creating the Trace Table
    - Exercise 31.4-2 Using a Non-Existing Index

- 31.5 How to Alter the Value of an Array Element
- 31.6 How to Iterate Through a One-Dimensional Array

    - Exercise 31.6-1 Finding the Sum

- 31.7 How to Add User-Entered Values to a One-Dimensional Array

    - Exercise 31.7-1 Displaying Words in Reverse Order
    - Exercise 31.7-2 Displaying Positive Numbers in Reverse Order
    - Exercise 31.7-3 Finding the Average Value
    - Exercise 31.7-4 Displaying Reals Only
    - Exercise 31.7-5 Displaying Elements with Odd-Numbered Indexes
    - Exercise 31.7-6 Displaying Even Numbers in Odd–Numbered Index Positions

- 31.8 What is a HashMap?
- 31.9 Creating HashMaps in Java
- 31.10 How to Get a Value from a HashMap

    - Exercise 31.10-1 Roman Numerals to Numbers
    - Exercise 31.10-2 Using a Non-Existing Key in HashMaps

- 31.11 How to Alter the Value of a HashMap Element

    - Exercise 31.11-1 Assigning a Value to a Non-Existing Key

- 31.12 How to Iterate Through a HashMap
- 31.13 Review Questions: True/False
- 31.14 Review Questions: Multiple Choice
- 31.15 Review Exercises

- Chapter 32 Two-Dimensional Arrays

  - 32.1 Creating Two-Dimensional Arrays in Java
  - 32.2 How to Get Values from Two-Dimensional Arrays

    - Exercise 32.2-1 Creating the Trace Table

  - 32.3 How to Iterate Through a Two-Dimensional Array
  - 32.4 How to Add User-Entered Values to a Two-Dimensional Array

    - Exercise 32.4-1 Displaying Reals Only
    - Exercise 32.4-2 Displaying Odd Columns Only

  - 32.5 What's the Story on Variables i and j?
  - 32.6 Square Matrices

    - Exercise 32.6-1 Finding the Sum of the Elements on the Main Diagonal
    - Exercise 32.6-2 Finding the Sum of the Elements on the Antidiagonal
    - Exercise 32.6-3 Filling in the Array

  - 32.7 Review Questions: True/False
  - 32.8 Review Questions: Multiple Choice
  - 32.9 Review Exercises

- Chapter 33 Tips and Tricks with Data Structures

  - 33.1 Introduction
  - 33.2 Processing Each Row Individually

    - Exercise 33.2-1 Finding the Average Value

- 33.3 Processing Each Column Individually
  - Exercise 33.3-1 Finding the Average Value
- 33.4 How to Use More Than One Data Structures in a Program
  - Exercise 33.4-1 Using Three One-Dimensional Arrays
  - Exercise 33.4-2 Using a One-Dimensional Array Along with a Two-Dimensional Array
  - Exercise 33.4-3 Using an Array Along with a HashMap
- 33.5 Creating a One-Dimensional Array from a Two-Dimensional Array
- 33.6 Creating a Two-Dimensional Array from a One-Dimensional Array
- 33.7 Useful Data Structures Methods (Subprograms), and More
- 33.8 Review Questions: True/False
- 33.9 Review Questions: Multiple Choice
- 33.10 Review Exercises

- Chapter 34 More with Data Structures
  - 34.1 Simple Exercises with Arrays
    - Exercise 34.1-1 Creating an Array that Contains the Average Values of its Neighboring Elements
    - Exercise 34.1-2 Creating an Array with the Greatest Values
    - Exercise 34.1-3 Merging One-Dimensional Arrays
    - Exercise 34.1-4 Creating Two Arrays – Separating Positive from Negative Values
    - Exercise 34.1-5 Creating an Array with Those who Contain Digit 5
  - 34.2 Data Validation with Arrays

    - Exercise 34.2-1 Displaying Odds in Reverse Order

- 34.3 Finding Minimum and Maximum Values in Arrays

    - Exercise 34.3-1 Which Depth is the Greatest?
    - Exercise 34.3-2 Which Lake is the Deepest?
    - Exercise 34.3-3 Which Lake, in Which Country, Having Which Average Area, is the Deepest?
    - Exercise 34.3-4 Which Students Have got the Greatest Grade?
    - Exercise 34.3-5 Finding the Minimum Value of a Two-Dimensional Array
    - Exercise 34.3-6 Finding the City with the Coldest Day
    - Exercise 34.3-7 Finding the Minimum and the Maximum Value of Each Row

- 34.4 Sorting Arrays

    - Exercise 34.4-1 The Bubble Sort Algorithm – Sorting One-Dimensional Arrays with Numeric Values
    - Exercise 34.4-2 Sorting One-Dimensional Arrays with Alphanumeric Values
    - Exercise 34.4-3 Sorting One-Dimensional Arrays While Preserving the Relationship with a Second Array
    - Exercise 34.4-4 Sorting Last and First Names
    - Exercise 34.4-5 Sorting a Two-Dimensional Array
    - Exercise 34.4-6 The Modified Bubble Sort Algorithm – Sorting One-Dimensional Arrays
    - Exercise 34.4-7 The Selection Sort Algorithm – Sorting One-Dimensional Arrays
    - Exercise 34.4-8 Sorting One-Dimensional Arrays While Preserving the Relationship with a Second Array
    - Exercise 34.4-9 The Insertion Sort Algorithm – Sorting One-Dimensional Arrays
    - Exercise 34.4-10 The Three Worst Elapsed Times

- 34.5 Searching Elements in Data Structures

- Exercise 34.5-1 The Linear Search Algorithm – Searching in a One-Dimensional Array that may Contain the Same Value Multiple Times
- Exercise 34.5-2 Display the Last Names of All Those People Who Have the Same First Name
- Exercise 34.5-3 The Linear Search Algorithm – Searching in a Two-Dimensional Array that May Contain the Same Value Multiple Times
- Exercise 34.5-4 The Linear Search Algorithm – Searching in a One-Dimensional Array that Contains Unique Values
- Exercise 34.5-5 Searching for a Social Security Number
- Exercise 34.5-6 The Linear Search Algorithm – Searching in a Two-Dimensional Array that Contains Unique Values
- Exercise 34.5-7 Checking if a Value Exists in all Columns
- Exercise 34.5-8 The Binary Search Algorithm – Searching in a Sorted One-Dimensional Array
- Exercise 34.5-9 Display all the Historical Events for a Country
- Exercise 34.5-10 Searching in Each Column of a Two-Dimensional Array

- 34.6 Exercises of a General Nature with Data Structures

  - Exercise 34.6-1 On Which Days was There a Possibility of Snow?
  - Exercise 34.6-2 Was There Any Possibility of Snow?
  - Exercise 34.6-3 In Which Cities was There a Possibility of Snow?
  - Exercise 34.6-4 Display from Highest to Lowest Grades by Student, and in Alphabetical Order
  - Exercise 34.6-5 Archery at the Summer Olympics
  - Exercise 34.6-6 The Five Best Scorers
  - Exercise 34.6-7 Counting the Frequency of Vowels

        - 34.7 Review Questions: True/False
        - 34.8 Review Exercises

    - Review in “Data Structures in Java”

        - Review Crossword Puzzle
        - Review Questions

- Part VII Subprograms

    - Chapter 35 Introduction to Subprograms

        - 35.1 What Exactly is a Subprogram?
        - 35.2 What is Procedural Programming?
        - 35.3 What is Modular Programming?
        - 35.4 Review Questions: True/False

    - Chapter 36 User-Defined Subprograms

        - 36.1 Subprograms that Return a Value
        - 36.2 How to Make a Call to a Method
        - 36.3 Subprograms that Return no Values
        - 36.4 How to Make a Call to a void Method
        - 36.5 Formal and Actual Arguments
        - 36.6 How Does a Method Execute?

            - Exercise 36.6-1 Back to Basics – Calculating the Sum of Two Numbers
            - Exercise 36.6-2 Calculating the Sum of Two Numbers Using Fewer Lines of Code!

        - 36.7 How Does a void Method Execute?

            - Exercise 36.7-1 Back to Basics – Displaying the Absolute Value of a Number

        - 36.8 Review Questions: True/False
        - 36.9 Review Exercises

- Chapter 37 Tips and Tricks with Subprograms
  - 37.1 Can Two Subprograms use Variables of the Same Name?
  - 37.2 Can a Subprogram Call Another Subprogram?
  - 37.3 Passing Arguments by Value and by Reference
  - 37.4 Returning an Array
  - 37.5 Overloading Methods
  - 37.6 The Scope of a Variable
  - 37.7 Converting Parts of Code into Subprograms
  - 37.8 Recursion
  - 37.9 Review Questions: True/False
  - 37.10 Review Exercises
- Chapter 38 More with Subprograms
  - 38.1 Simple Exercises with Subprograms
    - Exercise 38.1-1 A Simple Currency Converter
    - Exercise 38.1-2 Finding the Average Values of Positive Integers
    - Exercise 38.1-3 Finding the Sum of Odd Positive Integers
    - Exercise 38.1-4 Finding the Values of y
  - 38.2 Exercises of a General Nature with Subprograms
    - Exercise 38.2-1 Validating Data Input Using a Subprogram
    - Exercise 38.2-2 Sorting an Array Using a Subprogram
    - Exercise 38.2-3 Progressive Rates and Electricity Consumption
    - Exercise 38.2-4 Roll, Roll, Roll the… Dice!
    - Exercise 38.2-5 How Many Times Does Each Number of the Dice Appear?
  - 38.3 Review Exercises

    - Review in “Subprograms”
        - Review Crossword Puzzle
        - Review Questions
- Part VIII Object-Oriented Programming
    - Chapter 39 Introduction to Object-Oriented Programming
        - 39.1 What is Object-Oriented Programming?
        - 39.2 Classes and Objects in Java
        - 39.3 The Constructor and the Keyword this
        - 39.4 Passing Initial Values to the Constructor
            - Exercise 39.4-1 Historical Events
        - 39.5 Getter and Setter Methods
            - Exercise 39.5-1 The Roman Numerals
        - 39.6 Can a Method Call Another Method of the Same Class?
            - Exercise 39.6-1 Doing Math
        - 39.7 Class Inheritance
        - 39.8 Review Questions: True/False
        - 39.9 Review Exercises
    - Review in “Object-Oriented Programming”
        - Review Crossword Puzzle
        - Review Questions
- Part IX Files
    - Chapter 40 Introduction to Files
        - 40.1 Introduction
        - 40.2 Opening a File

        - 40.3 Closing a File
        - 40.4 Writing in (or Appending to) a File
        - 40.5 The File Pointer
        - 40.6 Reading from a File
        - 40.7 Iterating Through the Contents of a File
        - 40.8 Review Questions: True/False
        - 40.9 Review Exercises

    - Chapter 41 More with Files

        - 41.1 Exercises of a General Nature with Files

            - Exercise 41.1-1 Calculating the Sum of 10 Numbers
            - Exercise 41.1-2 Calculating the Average Value of an Unknown Quantity of Numbers
            - Exercise 41.1-3 Finding Minimum and Maximum Values
            - Exercise 41.1-4 Concatenating Files
            - Exercise 41.1-5 Searching in a File
            - Exercise 41.1-6 Combining Files with Subprograms

        - 41.2 Review Exercises

    - Review in “Files”

        - Review Crossword Puzzle
        - Review Questions

- Some Final Words from the Author

Java and Algorithmic Thinking for the Complete Beginner
Learn to Think Like a Programmer

3rd Revised Edition

By
Aristides S. Bouras

Java and Algorithmic Thinking for the Complete Beginner 3rd Revised Edition

Copyright © by Aristides S. Bouras https://www.bouraspage.com

Cover illustration: Philippos Papanikolaou Cover design: Muhammad Arslan

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

All crossword puzzles were created with EclipseCrossword software powered by Green Eclipse Python and PyCon are trademarks or registered trademarks of the Python Software Foundation.

PHP is a copyright of the PHP Group.

The following are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries: Microsoft, Windows, Visual Studio, Visual Studio Code, IntelliSense, SQL Server, VBA, Visual Basic, and Visual C#, .NET Framework.

Mazda and Mazda 6 are trademarks of the Mazda Motor Corporation or its affiliated companies.

Ford and Ford Focus are trademarks of the Ford Motor Company.

Other names may be trademarks of their respective owners.

Rcode: 240601

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, mechanical or electronic, including photocopying, recording, or by any information storage and retrieval system, without written permission from the author.

Warning and Disclaimer

This book is designed to provide information about learning “Algorithmic Thinking”, mainly through the use of Java programming language. Every effort has been taken to make this book compatible with the latest release of Java, and it is almost certain to be compatible with any future releases of it.

The information is provided on an “as is” basis. The authors shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the files that may accompany it.

Preface

About the Author

I was born in 1973, and from my early childhood, I discovered a passion for computer programming. At the age of 12, I got my first computer—a Commodore 64, which incorporated a ROM-based version of the BASIC programming language and 64 kilobytes of RAM. It introduced me to the world of programming!

I hold a degree in Computer Engineering from the Technological Educational Institute of Piraeus. Additionally, I earned Dipl. Eng. and Integrated Master's degrees in Electrical and Computer Engineering from the Democritus University of Thrace.

In my previous role as a software developer at a company specializing in industrial data flow and product labeling, my responsibilities included developing software applications for data terminals and PC software for data collection and storage on a Microsoft SQL Server®.

Over the years, I've designed various applications, including warehouse management systems and websites for diverse companies and organizations. Presently, I serve as a computer science teacher in a vocational high school, instructing courses on computer programming, computer networks, programming tools for the Internet/intranets, and databases.

I've authored several books, primarily focusing on algorithmic and computational thinking, utilizing languages such as Python, C#, Java, C++, PHP, and Visual Basic.

Outside of my professional pursuits, I am happily married and have the joy of raising two wonderful children.

Acknowledgments

Acknowledgments

I would like to thank, with particular gratefulness, my friend and senior editor Victoria (Vicki) Austin for her assistance in copy editing the first edition. Without her, this book might not have reached its full potential. With her patient guidance and valuable and constructive suggestions, she helped me bring this book up to a higher level!

How This Book is Organized

The book you hold in your hands follows the spiral curriculum teaching approach, a method proposed in 1960 by Jerome Bruner, an American psychologist. According to this method, as a subject is being taught, basic ideas are revisited at intervals—at a more sophisticated level each time—until the reader achieves a complete understanding of the subject. First, the reader learns the basic elements without worrying about the details. Later, more details are taught and basic elements are mentioned again and again, eventually being stored in the brain's long-term memory.

According to Jerome Bruner, learning requires the student's active participation, experimentation, exploration, and discovery. This book contains many examples, most of which can be practically performed. This gives the readers the opportunity to get their hands on Java® and become capable of creating their own programs.

Who Should Buy This Book?

Completely updated for the latest version of Java, this book offers a comprehensive introduction to programming, assuming no prior knowledge of the subject. It is designed for individuals, eager to learn Java from scratch, providing a strong foundation in Algorithmic Thinking—the fundamental skill every aspiring programmer must acquire. Algorithmic Thinking encompasses more than just writing code; it entails mastering the art of problem-solving through coding.

This edition retains all the popular features of its predecessor while introducing a wealth of new exercises, along with extensive revisions and

updates. Furthermore, brand-new chapters offer a practical introduction to working with text files.

Structured for both classroom use and independent study, each chapter is concluded with a set of questions and exercises designed to reinforce your understanding and apply what you've learned. With approximately 250 solved and 480 unsolved exercises, 500 true/false questions, 150 multiple-choice questions, and 200 review questions and crosswords (with solutions and answers available online), this book is ideal for:

- Novices or intermediate-level programmers pursuing self-study
- High school students
- First-years college or university students
- Educators
- Professors
- Anyone who wants to start learning or teaching computer programming using the best practices and techniques

Conventions Used in This Book

Following are some explanations on the conventions used in this book. "Conventions" refer to the standard ways in which certain parts of the text are displayed.

Java Statements This book uses plenty of examples written in Java language. Java statements are shown in a typeface that looks like this.

```
This is a Java statement
```

Keywords, Variables, Methods, and Arguments Within the Text of a Paragraph Keywords, variables, methods (subprograms), and arguments are sometimes shown within the text of a paragraph. When they are, the special text is shown in a typeface different from that of the rest of the paragraph. For instance, `firstName = 5` is an example of a Java statement within the paragraph text.

Words in Italics You may notice that some of the special text is also displayed in italics. In this book, italicized words are general types that must be replaced with the specific name appropriate for your data. For example, the general form of a Java statement may be presented as

static void *name*(***type1*** *arg1*, ***type2*** *arg2*)

In order to complete the statement, the keywords *`name`*, *`type1`*, *`arg1`*, *`type2`*, and *`arg2`* must be replaced with something meaningful. When you use this statement in your program, you might use it in the following form

static void displayRectangle(**int** width, **int** height)

Three dots (…): an Ellipsis In the general form of a statement you may also notice three dots (…), also known as an "ellipsis", following a list in an example. They are not part of the statement. An ellipsis indicates that you can have as many items in the list as you want. For example, the ellipsis in the general form of the statement

displayMessages(*arg1*, *arg2*, …);

indicates that the list may contain more than two arguments. When you use this statement in your program, your statement might be something like this.

```
displayMessages(message1, "Hello", message2, "Hi!");
```

Square Brackets in Italics The general form of some statements or methods (subprograms) may contain "square brackets" *`[ ]`* in italics,

which indicate that the enclosed section is optional. For example, the general form of the statement

subject.**substring**(*beginIndex [, endIndex]*)

indicates that the section *[, endIndex]* can be omitted.

For example, the following two statements may produce different results but they are both syntactically correct.

```
a = s.substring(3); b = s.substring(3, 9);
```

The Dark Header Most of this book's examples are shown in a typeface that looks like this.

Project_29.2-3

```
public static void main(String[] args) throws Exception {
   int a, b; a = 1; b = 2; System.out.println(a + b); }
```

The header **Project_29.2-3** on top indicates the filename that you must open to test the program. All the examples that contain this header can be downloaded free of charge from my website.

Notices Very often this book uses notices to help you better understand the meaning of a concept. Notices look like this.

This typeface designates a note.

Something Already Known or Something to Remember Very often this book can help you recall something you have already learned (probably in a previous section or chapter). Other times, it will draw your attention to something you should memorize. Reminders look like this.

This typeface designates something to recall or something that you should memorize.

How to Report Errata

Although I have taken great care to ensure the accuracy of the content in this book, mistakes can still occur. If you come across any errors, either in the text or the code, I highly encourage you to send me a report. By doing so, you'll not only assist in saving other readers from potential confusion and frustration but also contribute to enhancing the quality of the next

release. If you discover any errors, please report them by visiting one of the following addresses: https://tinyurl.com/28nwh2nf

https://www.bouraspage.com/report-errata

Once I verify your reported error(s), your submission will be accepted. The errata will then be uploaded to my website and added to any existing list of corrections.

Where to Download Material About this Book

Material about this book, such as: ► a list of verified errata (if any); ► the Solutions Companion, providing answers to all review questions and solutions to exercises; and ► all of this book's examples that have a header like this `Project_29.2-3` on top can be downloaded free of charge from the following address: https://tinyurl.com/3z44xhpp

https://www.bouraspage.com/books/java-and-algorithmic-thinking-for-the-complete-beginner-third-edition

If you Like this Book

If you find this book valuable, please consider visiting the web store where you purchased it, as well as goodreads.com, to show your appreciation by writing a positive review and awarding as many stars as you think appropriate. By doing so, you will motivate me to keep writing and, of course, you'll be assisting other readers in discovering my work.

Part I
Introductory Knowledge

Chapter 1
How a Computer Works

1.1 Introduction

In today's society, almost every task requires the use of a computer. In schools, students use computers to search the Internet and to send emails. At work, people use them to make presentations, to analyze data, and to communicate with customers. At home, people use computers to play games, to connect to social networks and to chat with other people all over the world. Of course, don't forget smartphones such as iPhones. They are computers as well!

Computers can perform so many different tasks because of their ability to be programmed. In other words, a computer can perform any job that a program tells it to. A *program* is a set of *statements* (often called *instructions* or *commands*) that a computer follows in order to perform a specific task.

Programs are essential to a computer, because without them a computer is a dummy machine that can do nothing at all. It is the program that actually tells the computer what to do and when to do it. On the other hand, the *programmer* or the *software developer* is the person who designs, creates, and often tests computer programs.

This book introduces you to the basic concepts of computer programming using the Java language.

1.2 What is Hardware?

The term *hardware* refers to all devices or components that make up a computer. If you have ever opened the case of a computer or a laptop you have probably seen many of its components, such as the microprocessor (CPU), the memory, and the hard disk. A computer is not a device but a system of devices that all work together. The basic components of a typical computer system are discussed here.

- **The Central Processing Unit (CPU)**

  This is the part of a computer that actually performs all the tasks defined in a program (basic arithmetic, logical, and input/output operations).

- **Main Memory (RAM – Random Access Memory)**

  This is the area where the computer holds the program (while it is being executed/run) as well as the data that the program is working with. All programs and data stored in this type of memory are lost when you shut down your computer or you unplug it from the wall outlet.

- **Main Memory (ROM – Read Only Memory)**

  ROM or Read Only Memory is a special type of memory which can only be *read* by the computer (but cannot be changed). All programs and data stored in this type of memory are **not** lost when the computer is switched off. ROM usually contains manufacturer's instructions as well as a program called the *bootstrap loader* whose function is to start the operation of computer system once the power is turned on.

- **Secondary Storage Devices**

  This is usually the hard disk or the SSD (Solid State Drive), and sometimes (but more rarely) the CD/DVD drive. In contrast to main memory (RAM), this type of memory can hold data for a longer period of time, even if there is no power to the computer. However, programs stored in this memory cannot be directly executed. They must be transferred to a much faster memory; that is, the main memory.

- **Input Devices**

  Input devices are all those devices that collect data from the outside world and enter them into the computer for further processing. Keyboards, mice, and microphones are all input devices.

- **Output Devices**

  Output devices are all those devices that output data to the outside world. Monitors (screens) and printers are output devices.

1.3 What is Software?

Everything that a computer does is controlled by software. There are two categories of software: system software and application software.

- *System software* is the program that controls and manages the basic operations of a computer. For example, system software controls the computer's internal operations. It manages all devices that are connected to it, and it saves data, loads data, and allows other programs to be executed. The three main types of system software are:
  - the *operating system*. Windows, Linux, macOS, Android, and iOS are all examples of operating systems.
  - the *utility software*. This type of software is usually installed with the operating system. It is used to make the computer run as efficiently as possible. Antivirus utilities and backup utilities are considered utility software.
  - the *device driver software*. A device driver controls a device that is attached to your computer, such as a mouse or a graphic card. A device driver is a program that acts like a translator. It translates the instructions of the operating system to instructions that a device can actually understand.
- *Application software* refers to all the other programs that you use for your everyday tasks, such as browsers, word processors, notepads, games, and many more.

1.4 How a Computer Executes (Runs) a Program

When you turn on your computer, the main memory (RAM) is completely empty. The first thing the computer needs to do is to transfer the operating system from the hard disk to the main memory.

After the operating system is loaded to main memory, you can execute (run) any program (application software) you like. This is usually done by clicking, double clicking, or tapping the program's corresponding icon. For example, let's say you click on the icon of your favorite word processor. This action orders your computer to copy (or load) the word processing program from your hard disk to the main memory (RAM) so the CPU can execute it.

Programs are stored on secondary storage devices such as hard disks. When you install a program on your computer, the program is copied to your hard disk. Then, when you execute a program, the program is copied (loaded) from your hard disk to the main memory (RAM), and that copy of the program is executed.

The terms "run" and "execute" are synonymous and can be used interchangeably.

1.5 Compilers and Interpreters

Computers can execute programs that are written in a strictly defined computer language. You cannot write a program using a natural language such as English or Greek, because your computer won't understand you!

But what does a computer actually understand? A computer can understand a specific low-level language called the *machine language*. In a machine language all statements (or commands) are made up of zeros and ones. The following is an example of a program written in a machine language, that calculates the sum of two numbers.

```
0010 0001 0000 0100
0001 0001 0000 0101
0011 0001 0000 0110
0111 0000 0000 0001
```

Shocked? Don't worry, you are not going to write programs this way. Hopefully, no one writes computer programs this way anymore. Nowadays, all programmers write their programs in a high-level language and then they use a special program to translate them into a machine language.

A high-level language is one that is not limited to a particular type of computer.

There are two types of programs that programmers use to perform translation: compilers and interpreters.

A *compiler* is a program that translates statements written in a high-level language into a separate machine language program. You can then execute the machine language program any time you wish. After the translation,

there is no need to run the compiler again unless you make changes in the high-level language program.

An *interpreter* is a program that simultaneously translates and executes the statements written in a high-level language. As the interpreter reads each individual statement in the high-level language program, it translates it into a machine language code and then directly executes it. This process is repeated for every statement in the program.

1.6 What is Source Code?

The statements (often called instructions or commands) that the programmer writes in a high-level language are called *source code* or simply *code*. The programmer first types the source code into a program known as a *code editor*, and then uses either a compiler to translate it into a machine language program, or an interpreter to translate and execute it at the same time.

> ✎ *While it may seem uncommon nowadays, it's entirely possible to write programs using a simple text editor!*

1.7 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Modern computers can perform so many different tasks because of their ability to be programmed.
2) A computer can operate without a program.
3) A hard disk is an example of hardware.
4) Data can be stored in main memory (RAM) for a long period of time, even if there is no power to the computer.
5) Data is stored in main memory (RAM), but programs are not.
6) Speakers are an example of an output device.
7) Windows and Linux are examples of software.
8) A device driver is an example of hardware.
9) A media player is an example of system software.
10) When you turn on your computer, the main memory (RAM) already contains the operating system.

11) When you open your word processing application, it is actually copied from a secondary storage device to the main memory (RAM).
12) In a machine language, all statements (commands) are a sequence of zeros and ones.
13) Nowadays, a computer cannot understand zeros and ones.
14) Nowadays, software is written in a language composed of ones and zeros.
15) Software refers to the physical components of a computer.
16) The compiler and the interpreter are software.
17) The compiler translates source code to an executable file.
18) The interpreter creates a machine language program.
19) Considering that a program might be executed multiple times, after it has been translated through interpretation and executed once, the need for the interpreter becomes obsolete.
20) Source code can be written using a simple text editor.
21) Source code can be executed by a computer without compilation or interpretation.
22) A program written in machine language requires compilation (translation).
23) A compiler translates a program written in a high-level language.

1.8 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) Which of the following is **not** computer hardware?
    a) a hard disk
    b) a DVD disc
    c) a sound card
    d) the main memory (RAM)
2) Which of the following is **not** a secondary storage device?
    a) a DVD reader/writer device
    b) a Solid State Drive (SSD)
    c) a USB flash drive

d) RAM

3) Which one of the following operations can**not** be performed by the CPU?
   a) Transfer data to the main memory (RAM).
   b) Transfer data from the main memory (RAM).
   c) Perform arithmetic operations.
   d) Surgical operations.
4) A touch screen is
   a) an input device.
   b) an output device.
   c) both of the above
5) Which of the following is **not** software?
   a) Windows
   b) Linux
   c) iOS
   d) a video game
   e) a web browser
   f) All of the above are software.
6) Which of the following statements is correct?
   a) Programs are stored on the hard disk.
   b) Programs are stored on USB flash drives (USB sticks).
   c) Programs are stored in main memory (RAM).
   d) All of the above are correct.
7) Which of the following statements is correct?
   a) Programs are executed directly from the hard disk.
   b) Programs are executed directly from a DVD disc.
   c) Programs are executed directly from the main memory (RAM).
   d) All of the above are correct.
   e) None of the above is correct.
8) Programmers can**not** write computer programs in

a) machine language.
b) natural language such as English, Greek, and so on.
c) Java.

9) A compiler translates
   a) a program written in machine language into a high-level language program.
   b) a program written in a natural language (English, Greek etc.) into a machine language program.
   c) a program written in high-level computer language into a machine language program.
   d) none of the above
   e) all of the above

10) Machine language is
   a) a language that machines use to communicate with each other.
   b) a language made up of numerical instructions that is used directly by a computer.
   c) a language that uses English words for operations.

11) In a program written in high-level computer language, if two identical statements are one after the other, the interpreter
   a) translates the first one and executes it, then it translates the second one and executes it.
   b) translates the first one, then translates the second one, and then executes them both.
   c) translates only the first one (since they are identical) and then executes it twice.

Chapter 2
Java and Integrated Development Environments

2.1 What is Java?

Java is a widely used general-purpose, high-level computer programming language that allows programmers to create desktop or mobile applications, web pages, and many other types of software. It is intended to let programmers “write once, run anywhere (WORA)”, meaning that code is written once but can run on any combination of hardware and operating system without being re-compiled.

2.2 What is the Difference Between a Script and a Program?

Technically speaking, a script is *interpreted* whereas a program is *compiled*, but this is actually not their major difference. There is another more important difference between them!

The main purpose of a script written in a scripting language such as JavaScript, or VBA (Visual Basic for Applications) is to control another application. So you can say that, in some ways JavaScript controls the web browser, and VBA controls a Microsoft® Office application such as MS Word or MS Excel.

On the other hand, a program written in a programming language such as Java, C++, or C# (to name a few) executes independently of any other application. A program is executed as stand-alone any time the user wishes without the need of a hosting application.

✎ *Macros of Microsoft Office are scripts written in VBA. Their purpose is to automate certain functions within Microsoft Office.*

✎ *A lot of people think that JavaScript is a simplified version of Java but in fact the similarity of the names is just a coincidence.*

A script cannot be executed as stand-alone. It requires a hosting application in order to execute.

2.3 Why You Should Learn Java

Java is what is known as a “high-level” computer language. Java's coding style is similar to C language. It is quite easy to understand and highly efficient on multiple platforms such as Windows, Android, Linux, and Unix.

Java is a flexible and powerful language, making it well-suited for developing games, web applications, desktop or mobile applications.

Java is everywhere! It is on desktop computers, laptops, datacenters, mobile devices, and game consoles. More than 9 billion mobile phones, most computers, and more than 125 million TV devices run Java. With more than 9 million programmers worldwide, Java enables efficient development of many exciting applications and services. This huge availability of Java programmers is a major reason why organizations choose Java for new development over any other programming language. This is also a very good reason why you should actually learn Java!

✎ *Android Apps are mostly written in Java!*

2.4 How Java Works

Computers do not understand natural languages such as English or Greek, so you need a computer language such as Java to communicate with them. Java is a very powerful high-level computer language. The Java interpreter (or, actually, a combination of a compiler and an interpreter) converts Java language to a language that computers can actually understand, and that is known as the “machine language”.

In the past, computer languages made use of either an interpreter or a compiler. Nowadays however, many computer languages including Java use both a compiler and an interpreter. The Java compiler translates Java statements into *bytecode* statements and saves them in a .class file. Later, when a user wants to execute the file, the Java Virtual Machine (JVM)—which is actually a combination of a compiler and an interpreter—reads the .class file and executes it, initially using interpretation. During interpretation, however, the JVM monitors which sequences of bytecode are frequently executed and translates them (compiles them) into low-level machine language code for direct execution on the hardware.

✎ *Java bytecode is a machine language executed by the Java Virtual Machine (JVM).*

✎ *Instead of a compiler and an interpreter, some languages use two compilers. In C#, for example, the first compiler translates C# statements into an intermediate language called Common Intermediate Language (CIL). The CIL code is stored on disk in an executable file called an assembly, typically with an extension of .exe. Later, when a user wants to execute the file, the .NET Framework performs a Just In Time (JIT) compilation to convert the CIL code into low-level machine language code for direct execution on the hardware.*

In **Figure 2–1** you can see how statements written in Java are compiled into bytecode and how bytecode is then executed using the Java Virtual Machine (JVM).

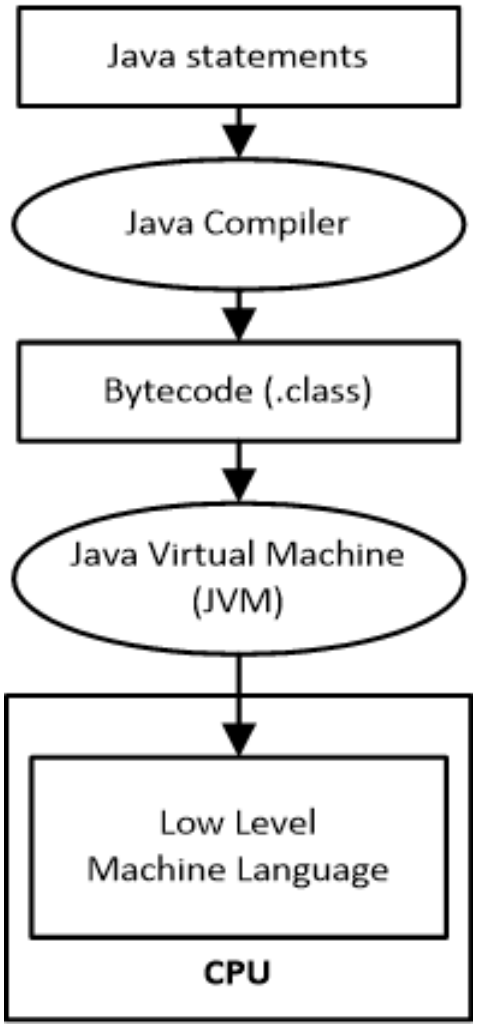


Figure 2–1 Executing Java statements using the Java Virtual Machine

Now come some reasonable questions: *Why all this trouble? Why does Java and other languages (such as C#) translate twice? Why are Java statements*

not directly translated into low-level machine language code? The answer lies in the fact that Java is designed to be a platform-independent programming language. This means that a program is written once but it can be executed on any device, regardless of its operating system or its architecture, as long as the appropriate version of Java is installed on it. In the past, programs had to be recompiled, or even rewritten, for each computer platform. One of the biggest advantages of Java is that you only have to write and compile a program once! In **Figure 2–2** you can see how statements written in Java are compiled into bytecode and how bytecode can then be executed on any platform that has the corresponding Java Virtual Machine (JVM) installed on it.

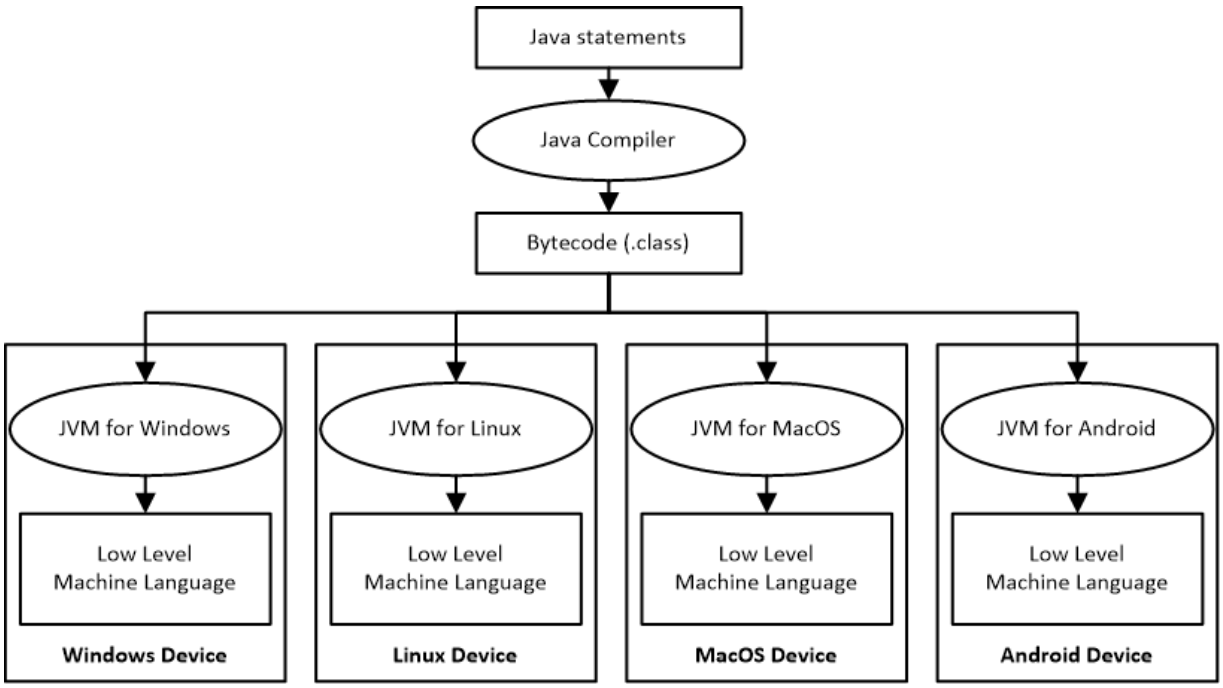


Figure 2–2 Executing Java statements on different platforms

✎ *Some platforms offer direct hardware support for Java; there are microcontrollers that can run Java directly on the hardware rather than using the software Java Virtual Machine. Many mobile phones use ARM-based processors (CPUs) which have hardware support for directly executing Java bytecode.*

To write and execute Java programs, you need to install two software applications: the Java Development Kit (JDK) and an Integrated Development Environment (IDE). The IDE is necessary for writing Java programs, while JDK is required for executing and debugging them.

2.5 Java Development Kit (JDK)

The Java Development Kit (JDK) is a free-of-charge development kit used for developing games, desktop, mobile or web applications using the Java programming language. The JDK provides everything needed to create, compile, and run Java applications, including the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), and other tools needed for Java development.

JDK is an essential tool for Java developers and can be installed on different platforms such as Windows, Linux, and macOS.

2.6 Integrated Development Environments

An Integrated Development Environment, or IDE, is a type of software that includes all the basic tools programmers need to write and test programs. An IDE typically contains a source code editor and integrates tools such as a compiler or an interpreter, along with a debugger. Visual Studio Code is an example of IDE that lets programmers write, execute and debug their source code.

> ✎ *A "debugger" is a tool that helps programmers to find and correct many of their mistakes.*

2.7 Microsoft Visual Studio

Microsoft Visual Studio is an Integrated Development Environment (IDE) that provides a great set of tools for many programming languages (via extensions installed separately) and lets you easily create applications for Android, iOS, macOS, Windows, and the cloud, as well as websites, web applications, and web services.

Visual Studio is much more than a text editor. It can indent lines, match words and brackets, and highlight source code that is written incorrectly. It also provides automatic code (IntelliSense®), which means that as you type, it displays a list of possible completions. The IDE also provides hints to help you analyze your code and find any potential problems. It even suggests some simple solutions to fix those problems. You can use the Visual Studio not only to write but also to execute and debug your programs directly from the IDE.

Visual Studio has a large community of users all around the world and this is why it comes in so many different flavors. Specifically, in Microsoft's

download page you can download:

- **Visual Studio** (Community, Professional, or Enterprise), which runs on Windows
- **Visual Studio for Mac**, which runs on macOS
- **Visual Studio Code**, which runs on Windows, macOS, and Linux

In the next chapter (Chapter 3), you will find links guiding you to instructions on how to install and configure whatever is necessary on your computer, such as the JDK and Visual Studio Code, on either Windows or Linux. Then, in Chapter 9, you will discover guidance on using Visual Studio Code to write, execute and debug Java programs. These instructions are available on my website. Additionally, you will find numerous tips and tricks there that will be valuable in your first steps as a budding programmer!

Chapter 3
Software Packages to Install

3.1 What to Install

For the purposes of this book, you need to install Java Development Kit (JDK) and an Integrated Development Environment (IDE) such as Visual Studio Code on your computer.

All the instructions you need regarding how to set up JDK and Visual Studio Code, on either Windows or Linux are maintained on my website at the following addresses. This gives me the flexibility to review them frequently and keep them up-to-date.

https://tinyurl.com/mvsa5mzn

https://www.bouraspage.com/java-setup-write-execute-debug

If you find any inconsistencies, please let me know, and I will update the instructions as soon as possible. To report issues, visit one of the following addresses: https://tinyurl.com/28nwh2nf

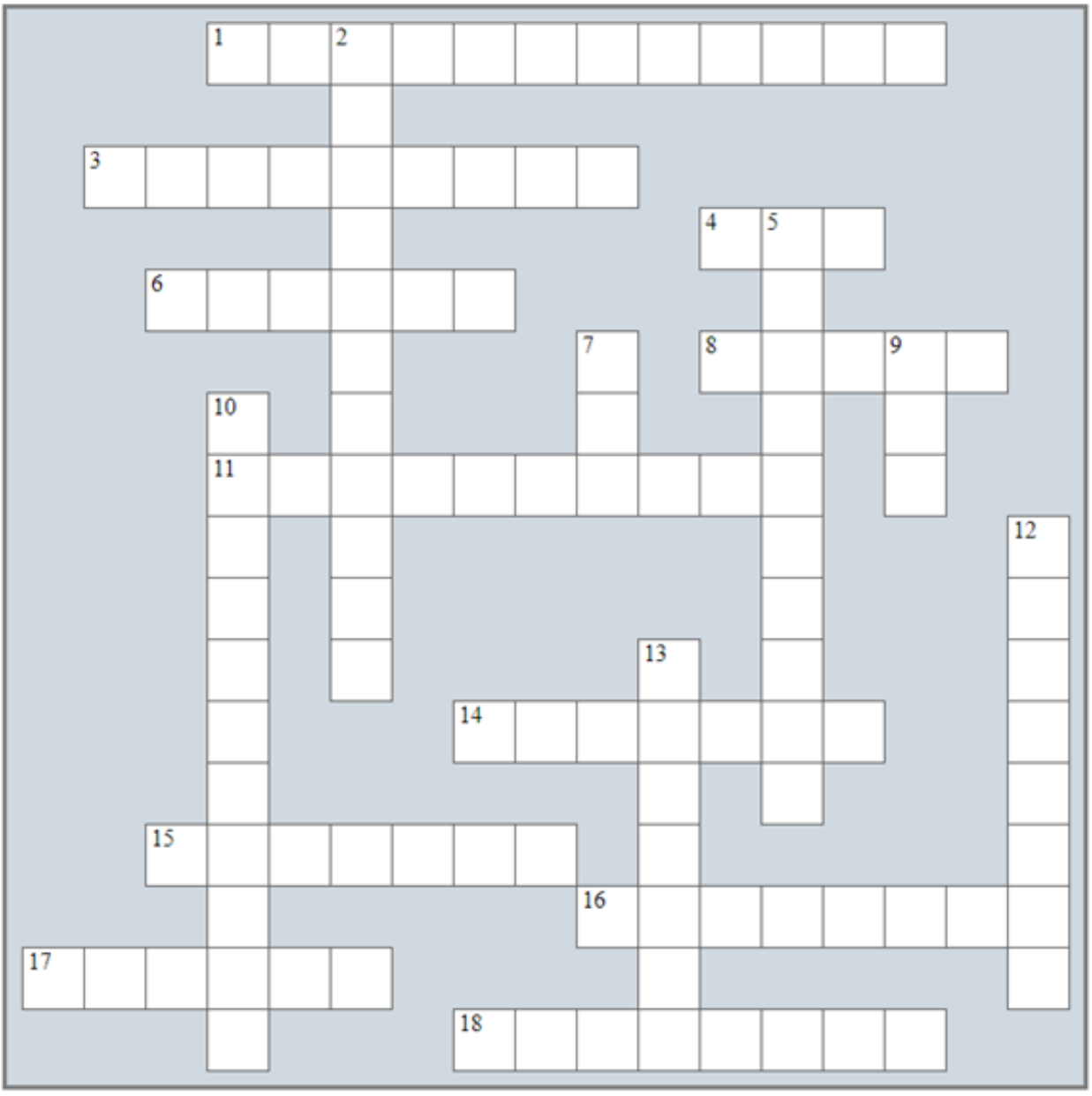
https://www.bouraspage.com/report-errata

✎ *Currently, all you need is to install JDK and an IDE. Instructions on my website regarding how to write, execute, and debug a Java program are unnecessary at this stage. You will require these instructions when you reach Chapter 9.*

Review in “Introductory Knowledge”

Review Crossword Puzzles

1) Solve the following crossword puzzle.



Across 1) Statements or commands.

3) Windows is such a system.

4) A computer component.

6) A category of software.

8) An input device.

11) It's the person who designs computer programs.

14) An output device.

15) Antivirus is such a software.

16) In today's society, almost every task requires the use of this device.

17) A computer component.

18) All these devices make up a computer.

Down 2) These devices are also computers.

5) Computers can perform so many different tasks because of their ability to be _____.

7) Special memory that can only be read.

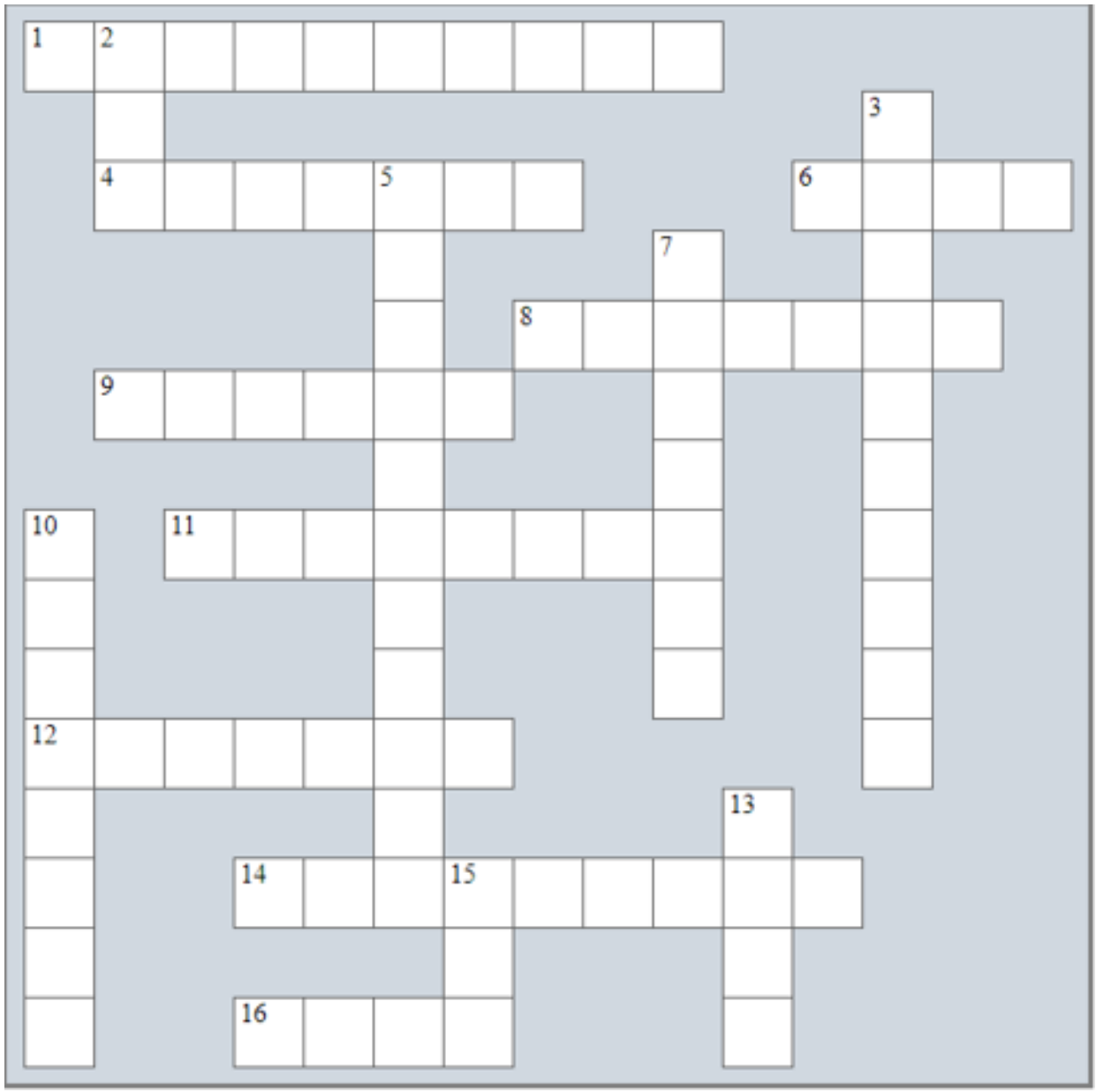
9) A secondary storage device.

10) A browser is this type of software.

12) An input device.

13) An operating system.

2) Solve the following crossword puzzle.



Across 1) The CPU performs one of these basic operations.

4) A low-level language.

6) Widely used general-purpose, high-level computer programming language.

8) Java _____ Machine converts the bytecode into low-level machine language code.

9) This software controls a device that is attached to your computer.

11) A program that translates statements written in a high-level language into a separate machine language program.

12) Run a program.

14) A category of programming language.

16) The statements that a programmer writes to solve a problem.

Down 2) All data stored in this type of memory are lost when you shut down your computer.

3) A scripting language.

5) A program that simultaneously translates and executes the statements written in a high-level language.

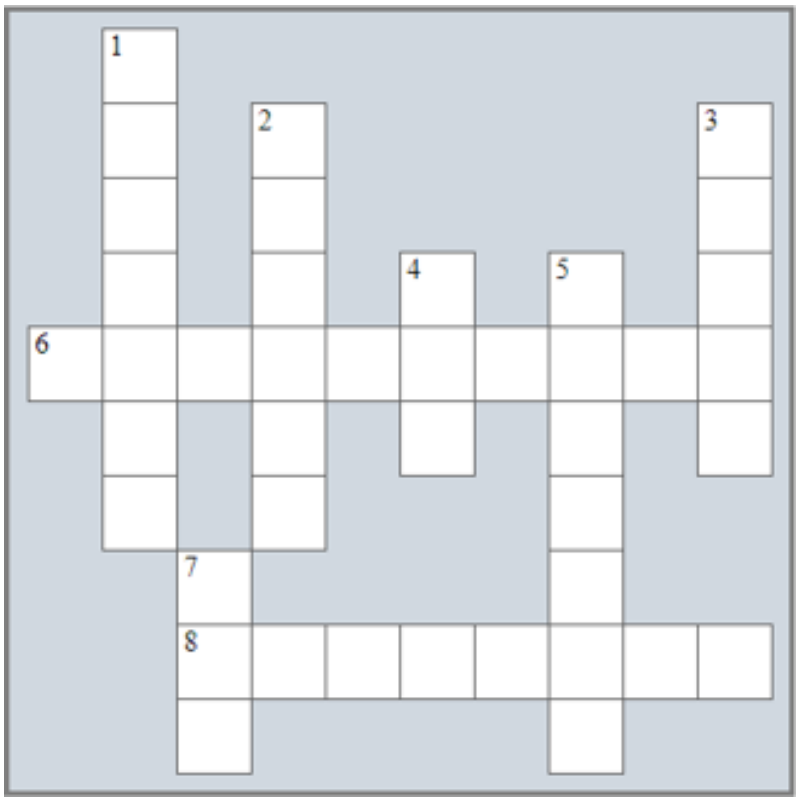
7) A set of statements.

10) Java _____ is a machine language executed by the Java Virtual Machine.

13) In a machine language all statements are made up of zeros and _____.

15) Visual Studio Code is such a software.

3) Solve the following crossword puzzle.



Across 6) An input device.

8) An intermediate language that Java uses.

Down 1) A script requires a _____ application in order to execute.

2) Scripts written in VBA.

3) Java is suitable for developing _____.

4) It performs logical operations.

5) It displays data to the user.

7) A scripting language that can control Microsoft Word.

Review Questions

Answer the following questions.

1) What is hardware?
2) List the five basic components of a typical computer system.
3) What does the “bootstrap loader” program do?
4) Which part of the computer actually executes the programs?
5) Which part of the computer holds the program and its data while the program is running?
6) Which part of the computer holds data for a long period of time, even when there is no power to the computer?
7) How do you call the device that collects data from the outside world and enters them into the computer?
8) List some examples of input devices.
9) How do you call the device that outputs data from the computer to the outside world?
10) List some examples of output devices.
11) What is software?
12) How many software categories are there, and what are their names?
13) A word processing program belongs to what category of software?
14) What is a compiler?
15) What is an interpreter?

16) What is meant by the term “machine language”?
17) What is source code?
18) What is Java?
19) What is bytecode?
20) What does the acronym JVM stand for?
21) What is the difference between a script and a program?
22) What are some of the possible uses of Java?
23) What is JDK?
24) What does the acronym JDK stand for?
25) What is Visual Studio Code?

Part II
Getting Started with Java

Chapter 4
Introduction to Basic Algorithmic Concepts

4.1 What is an Algorithm?

In technical terms, an *algorithm*[1] is a strictly defined finite sequence of well-defined statements (often called instructions or commands) that provides the solution to a problem or to a specific class of problems for any acceptable set of input values (if there are any inputs). In other words, an algorithm is a step-by-step procedure to solve a given problem. The term *finite* means that the algorithm must reach an end point and cannot run forever.

You can find algorithms everywhere in your real life, not just in computer science. For instance, the process of preparing toast or a cup of tea can be expressed as an algorithm. Certain steps, in a particular order, must be followed to achieve your goal.

4.2 The Algorithm for Making a Cup of Tea

The following is an algorithm for making a cup of tea.

1) Put a teabag in a cup.
2) Fill a kettle with water.
3) Boil the water in the kettle.
4) Pour some of the boiled water into the cup.
5) Add milk to the cup.
6) Add sugar to the cup.
7) Stir the tea.
8) Drink the tea.

As you can see, certain steps must be followed. These steps are in a specific order, even though some of the steps could be rearranged. For example, steps 5 and 6 can be reversed. You could add the sugar first, and the milk afterwards.

✎ *Keep in mind that the order of some steps can probably be changed but you can't move them far away from where they should be. For example, you can't move step 3 (“Boil the water in the kettle.”) to the end of the algorithm, because you will end up drinking a cup of iced tea (and not a warm one) which is totally different from your initial goal!*

4.3 Properties of an Algorithm

In his book *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Donald E. Knuth[3] asserts that an algorithm must satisfy the following five properties:

- **Input**: The algorithm must have input values from a specified set.
- **Output**: The algorithm must produce the output values from a specified set of input values. The output values are the solution to a problem.
- **Finiteness**: For any input, the algorithm must terminate after a finite number of steps.
- **Definiteness**: All steps of the algorithm must be precisely defined. Every instruction within the algorithm should be clear and unambiguous. An algorithm must explicitly describe how the computation is to be carried out. The property of definiteness ensures that the agent executing the instructions will always know which command to perform next. Some examples of algorithms that do not satisfy the property of definiteness are:
  - an algorithm that involves dividing a number by zero without any checks or safeguards. Dividing by zero is mathematically undefined, and an algorithm that doesn't handle this scenario can lead to unexpected results or errors in the computation.
  - an algorithm that attempts to calculate the square root of a negative number without accounting for complex numbers. The square root of a negative number is not a real number but a complex one. If the algorithm doesn't handle this properly, it might produce invalid or nonsensical results.
- **Effectiveness**: It refers to the ability of an algorithm to consistently and accurately produce a meaningful and correct result for all possible valid inputs (including edge cases) within a finite amount of time. The steps

of the algorithm must be basic enough so that, for example, someone using a pencil and paper could carry them out exactly.

4.4 Okay About Algorithms. But What is a Computer Program Anyway?

A *computer program* is nothing more than an algorithm that is written in a language that computers can understand, like Java, Python, C++, or C#.

A computer program cannot actually *make* you a cup of tea or cook your dinner, although an algorithm can guide you through the steps to do it yourself. However, programs can (for example) be used to calculate the average value of a set of numbers, or to find the maximum value among them. Artificial intelligence programs can even play chess or solve logic puzzles.

4.5 The Three Parties!

There are always three parties involved in an algorithm—the one that writes the algorithm, the one that executes it, and the one that uses or enjoys it.

Let's take an algorithm for preparing a meal, for example. Someone writes the algorithm (the author of the recipe book), someone executes it (probably your mother, who prepares the meal following the steps from the recipe book), and someone uses it (probably you, who enjoys the meal).

Now consider a real computer program. Let's take a video game, for example. Someone writes the algorithm in a computer language (the programmer), something executes it (usually a laptop or a computer), and someone uses it or plays with it (the user).

Be cautious, as sometimes the terms “programmer” and “user” can be a source of ambiguity. When you *write* a computer program, you temporarily assume the role of “the programmer” but when you *use* your own program, you take on the role of “the user”.

4.6 The Three Main Stages Involved in Creating an Algorithm

An algorithm should consist of three stages: *data input*, *data processing*, and *results output*. This order is specific and cannot be changed.

Consider a computer program that finds the average value of three numbers. First, the program must prompt (ask) the user to enter the numbers (the data input stage). Next, the program must calculate the average value of the

numbers (the data processing stage). Finally, the program must display the result on the computer's screen (the results output stage).

Let's take a look at these stages in more detail.

First stage – Data input

1) Prompt the user to enter a number.
2) Prompt the user to enter a second number.
3) Prompt the user to enter a third number.

Second stage – Data processing

4) Calculate the sum of the three numbers.
5) Divide the sum by 3.

Third stage – Results output

6) Display the result on the screen.

In some rare situations, the input stage may be absent and the computer program may consist of only two stages. For example, consider a computer program that is written to calculate the following sum.

$$1 + 2 + 3 + 4 + 5$$

In this example, the user must enter no values at all because the computer program knows exactly what to do. It must calculate the sum of the numbers 1 to 5 and then display the value of 15 on the user's screen. The two required stages (data processing and results output) are shown here.

First stage – Data input

Nothing to do

Second stage – Data processing

1) Calculate the sum of 1 + 2 + 3 + 4 + 5.

Third stage – Results output

2) Display the result on the screen.

However, what if you want to let the user decide the upper limit of that sum? What if you want to let the user decide whether to sum the numbers 1 to 5 or the numbers 1 to 20? In that case, the program must include an input stage at the beginning of the program to let the user enter that upper limit. Once the user enters that upper limit, the computer can calculate the result. The three required stages are shown here.

First stage – Data input

1) Prompt the user to enter a number.

Second stage – Data processing

2) Calculate the sum 1 + 2 + … (up to and including the upper limit the user entered).

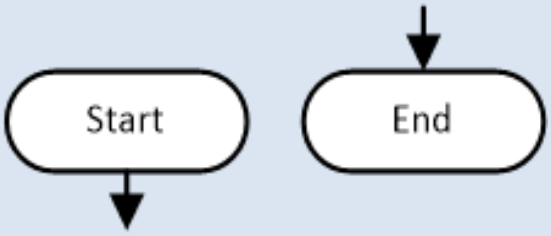
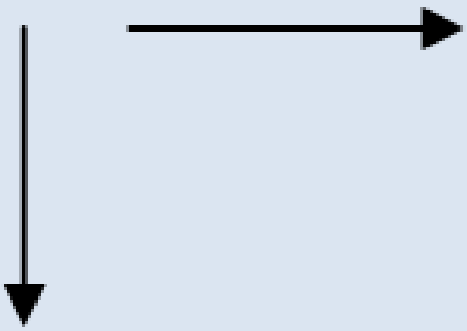
Third stage – Results output

3) Display the results on the screen.

For example, if the user enters the number 6 as the upper limit, the computer would find the result of 1 + 2 + 3 + 4 + 5 + 6.

4.7 Flowcharts

A *flowchart* is a graphical method of presenting an algorithm, usually on paper. It is the visual representation of the algorithm's flow of execution. In other words, it visually represents how the flow of execution proceeds from one statement to the next until the end of the algorithm is reached. The basic symbols that flowcharts use are shown in **Table 4-1**.

Flowchart Symbols	Description
	**Start/End**: Represents the beginning or the end of an algorithm. The Start symbol has one exit and the End symbol has one entrance.
	**Arrow**: Shows the flow of execution. An arrow coming from one symbol and ending at another symbol shows that control passes to the symbol that the arrow is pointing to. Arrows are always drawn as straight lines going up and down or sideways (never at an angle).
	Process: Represents a process or mathematical (formula) calculation. The Process

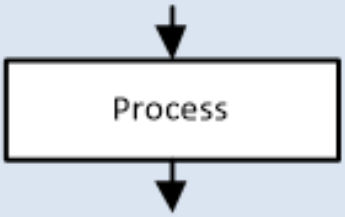
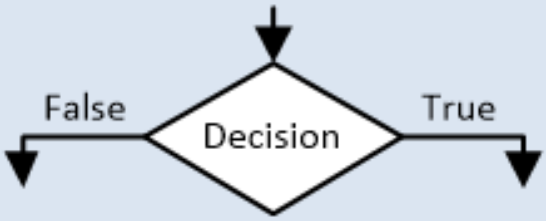
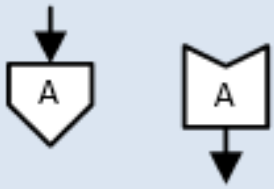
	symbol has one entrance and one exit.
	Data Input/Output: Represents the data input or the results output. In most cases, data comes from a keyboard and results are displayed on a screen. The Data input/output symbol has one entrance and one exit.
	**Decision**: Indicates the point at which a decision is made. Based on a given condition (which can be true or false), the algorithm will follow either the right or the left path. The Decision symbol has one entrance and two (and always only two) exits.
	**Off-page connectors**: Show continuation of a flowchart onto another page. They are used to connect segments on multiple pages when a flowchart gets too big to fit onto one sheet of paper. The outgoing off-page connector symbol has one entrance and the incoming off-page connector symbol has one exit.

Table 4-1 Flowchart symbols and their functions

An example of a flowchart is shown in **Figure 4–1.** The algorithm prompts the user to enter three numbers and then calculates their average value and

displays it on the computer screen.

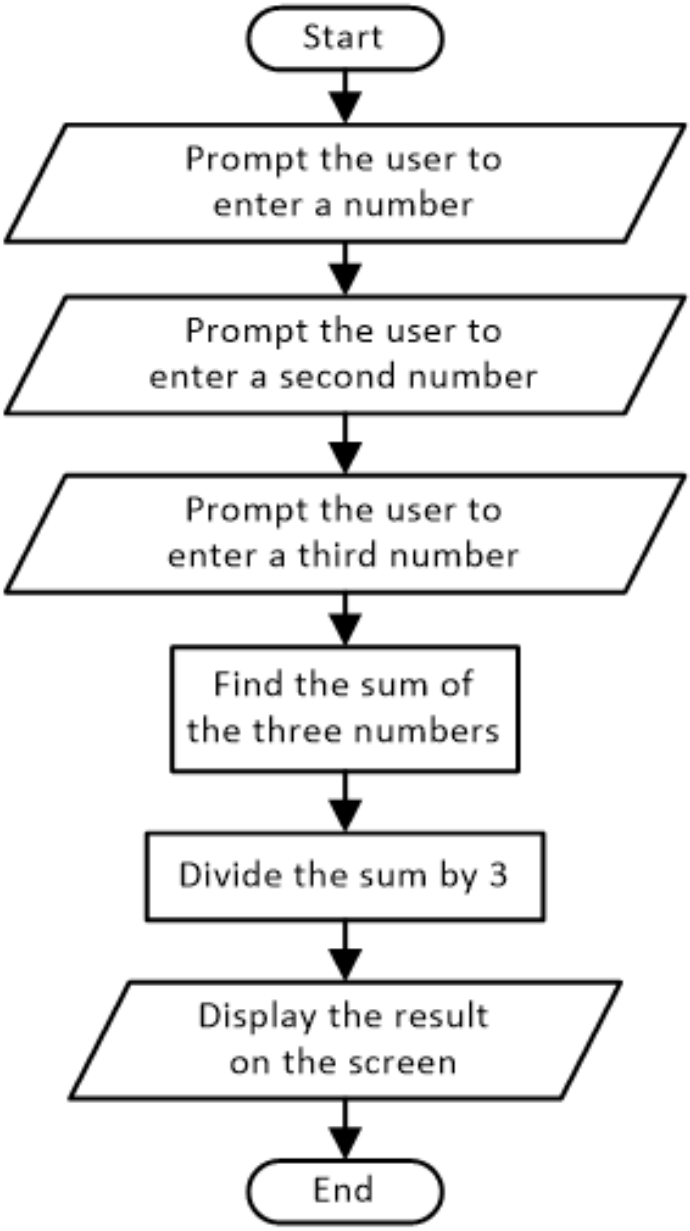

Start
Prompt the user to enter a number
Prompt the user to enter a second number
Prompt the user to enter a third number
Find the sum of the three numbers
Divide the sum by 3
Display the result on the screen
End

Figure 4–1 Flowchart for an algorithm that calculates and displays the average of three numbers

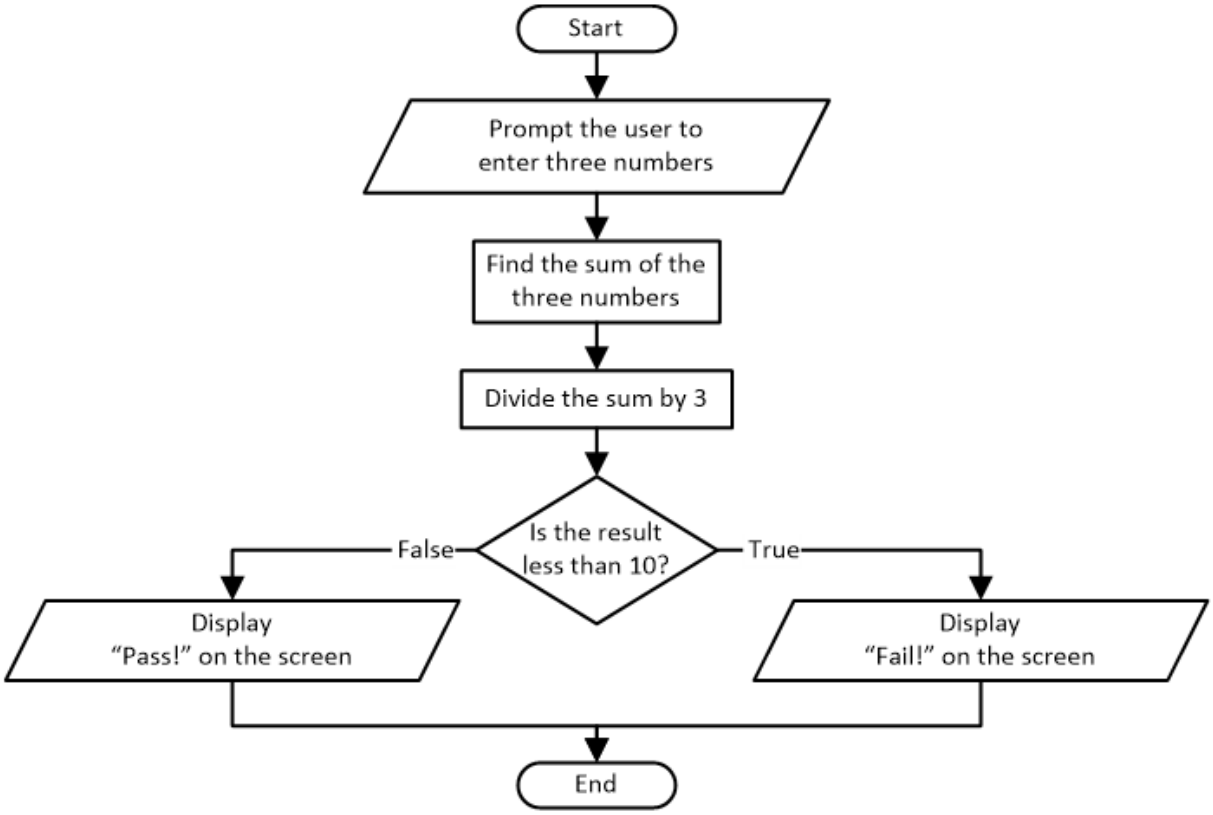
A flowchart always begins and ends with a Start/End symbol!

Exercise 4.7-1 Finding the Average Value of Three Numbers

Design an algorithm that calculates the average value of three numbers. Whenever the average value is below 10, a message "Fail!" must be displayed. Otherwise, if the average value is 10 or above, a message "Pass!" must be displayed.

Solution

In this problem, two different messages must be displayed, but only one can appear each time the algorithm is executed; the wording of the message depends on the average value. The flowchart for the algorithm is presented here.



To save paper, you can prompt the user to enter all three numbers using one single oblique parallelogram.

A Decision symbol always has one entrance and two exit paths!

Of course it is very soon for you to start creating your own algorithms. This particular exercise is quite simple and is presented in this chapter as an exception, just for demonstration purposes. You need to learn more before you start creating your own algorithms or even Java programs. Just be patient! In a few chapters the big moment will come!

4.8 What are ”Reserved Words”?

In a computer language, a *reserved word* (or *keyword*) is a word that has a strictly predefined meaning—it is reserved for special use and cannot be

used for any other purpose. For example, the words `Start`, `End`, `Read`, and `Write` in flowcharts have a predefined meaning. They are used to represent the beginning, the end, the data input, and the results output, respectively.

Reserved words exist in all high-level computer languages. In Java, there are many reserved words such as `if`, `while`, `else`, and `for`. However, each language has its own set of reserved words. For example, the reserved words `else if` in Java are written as `elif` in Python.

4.9 What is the Difference Between a Statement and a Command?

There is a big discussion on the Internet about whether there is, or is not, any difference between a statement and a command. Some people prefer to use the term “statement”, and some others the term “command”. For a novice programmer, there is no difference; both are instructions to the computer!

4.10 What is Structured Programming?

Structured programming is a software development method that uses modularization and structured design. Large programs are broken down into smaller modules and each individual module uses structured code, which means that the statements are organized in a specific manner that minimizes errors and misinterpretation. As its name suggests, structured programming is done in a structured programming language and Java is one such language.

The structured programming concept was formalized in 1966 by Corrado Böhm[4] and Giuseppe Jacopini[5]. They demonstrated theoretical computer program design using sequences, decisions, and iterations.

4.11 The Three Fundamental Control Structures

There are three fundamental control structures in structured programming.

- **Sequence Control Structure**: This refers to the line-by-line execution, in which statements are executed sequentially, in the same order in which they appear in the program, without skipping any of them. It is also known as a *sequential control structure*.
- **Decision Control Structure**: Depending on whether a condition is true or false, the decision control structure may skip the execution of an entire block of statements or even execute one block of statements instead of another. It is also known as a *selection control structure*.

- **Loop Control Structure**: This is a control structure that allows the execution of a block of statements multiple times until a specified condition is met. It is also known as an *iteration control structure* or a *repetition control structure*.

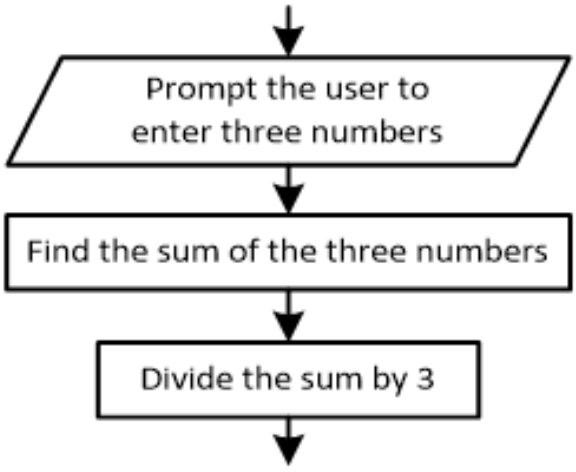
> ✎ *Every computer program around the world is written in terms of only these three control structures!*

Exercise 4.11-1 Understanding Control Structures Using Flowcharts

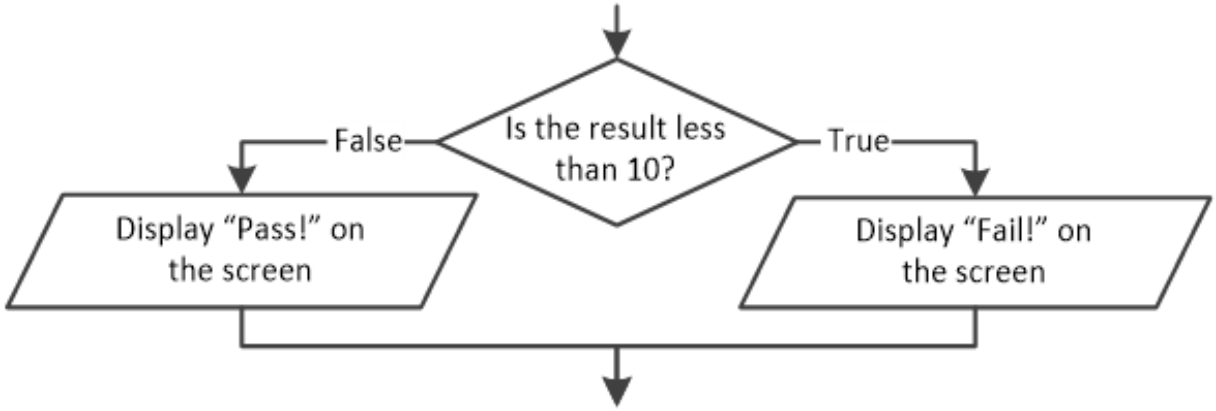
Using flowcharts, give an example for each type of control structure.

Solution

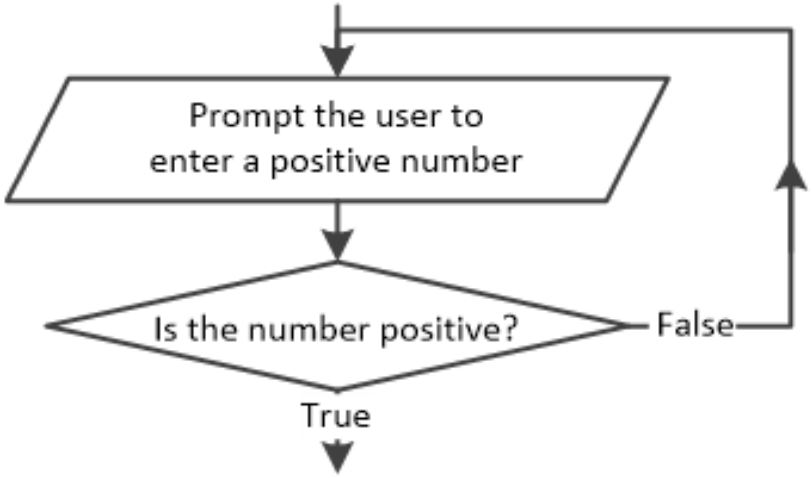
Example of a Sequence Control Structure



Example of a Decision Control Structure



Example of a Loop Control Structure



If you didn't quite understand the deeper meaning of these three control structures, don't worry, because upcoming chapters will analyze them very thoroughly. Patience is a virtue. All you have to do for now is wait!

4.12 Your First Java Program

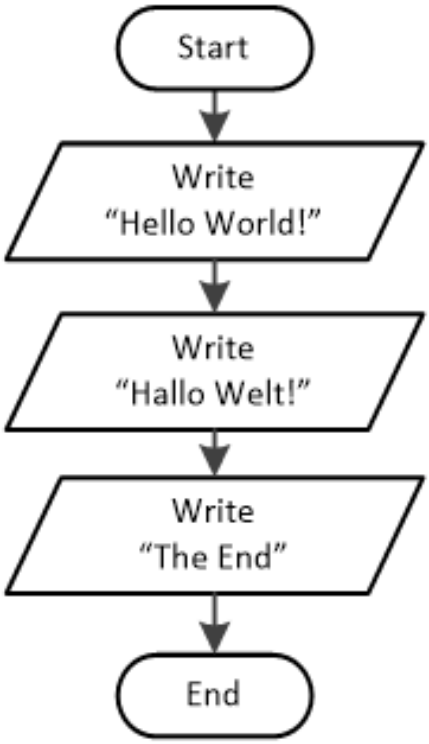
Converting a flowchart to a computer language such as Java results in a Java program. A Java program is nothing more than a text file including Java statements. Java programs can even be written in your text editor application! Keep in mind, though, that using Visual Studio Code to write Java programs is a much better solution due to all of its included features that can make your life easier.

> *A Java source code is saved on your hard disk with the default .java file extension.*

A Java program must always contain a `main` method, as shown here.

```
public static void main(String[] args) throws Exception {
   // Java code goes here
}
```

Here is a very simple algorithm that displays three messages on the screen.



And here is the same algorithm written as a Java program.

```
public static void main(String[] args) throws Exception {
   System.out.println("Hello World!");
   System.out.println("Hallo Welt!");
   System.out.println("The End");
}
```

✎ *Note that Java requires that all statements be terminated with a semicolon.*

4.13 What is the Difference Between a Syntax Error, a Logic Error, and a Runtime Error?

When high-level language code is written or executed, three types of errors may occur: syntax errors, logic errors, and runtime errors.

A *syntax error* is a mistake such as a misspelled keyword, a missing punctuation character, or a missing closing bracket. The syntax errors are detected by the compiler or the interpreter. If you try to execute a Java program that contains a syntax error, you will get an error message on your screen and the program won't execute. You must correct any errors and then try to execute the program again.

✎ *Some IDEs, such as Visual Studio Code, detect these errors as you type and underline the erroneous statements with a wavy red line.*

A *logic error* is an error that prevents your program from doing what you expected it to do. With logic errors you get no warning at all. Your code compiles and runs but the result is not the expected one. Logic errors are hard to detect. You must review your program thoroughly to find out where your error is. For example, consider a Java program that prompts the user to enter three numbers, and then calculates and displays their average value. In this program, however, the programmer made a typographical error (a "typo"); one of their statements divides the sum of the three numbers by 5, and not by 3 as it should. Of course the Java program executes as normal, without any error messages, prompting the user to enter three numbers and displaying a result, but obviously not the correct one! It is the programmer's responsibility to find and correct the erroneously written Java statement, not the computer, the interpreter or the compiler! Computers are not that smart after all!

A *runtime error* is an error that occurs during the execution of a program. A runtime error can cause a program to end abruptly or even cause system shut-down. Such errors are the most difficult errors to detect. There is no way to be sure, before executing the program, whether this error is going to happen, or not. You can suspect that it may happen though! For example, running out of memory or a division by zero causes a runtime error.

✎ *A logic error can be the cause of a runtime error!*

✎ *Logic errors and runtime errors are commonly referred to as "bugs", and are often found during the debugging process, before the software is released. When errors are found after a software has been released to the public, programmers often release patches, or small updates, to fix the errors.*

4.14 What "Debugging" Means

Debugging is the process of finding and reducing the number of defects (bugs) in a computer program, in order to make it perform as expected.

There is a myth about the origin of the term "debugging". In 1940, while Grace Hopper[6] was working on a Mark II Computer at Harvard University, her associates discovered a bug (a moth) stuck in a relay (an electrically

operated switch). This bug was blocking the proper operation of the Mark II computer. So, while her associates where trying to remove the bug, Grace Hopper remarked that they were “debugging” the system!

4.15 Commenting Your Code

When you write a small and easy program, anyone can understand how it works just by reading it line-by-line. However, long programs are difficult to understand, sometimes even by the same person who wrote them.

Comments are extra information that can be included in a program to make it easier to read and understand. Using comments, you can add explanations and other pieces of information, including:

- who wrote the program
- when the program was created or last modified
- what the program does
- how the program works

> ✎ *Comments are for human readers. Compilers and interpreters ignore any comments you may add to your programs.*

However, you should not over-comment. There is no need to explain every line of your program. Add comments only when a particular portion of your program is hard to follow.

In Java, you can add comments using one of the following methods:

- double slashes `//……`
- slash–asterisk, asterisk–slash delimiters `/*` …… `*/`

The following program demonstrates how to use both types of commenting. Usually double slashes (`//`) are used for commenting one single line, whereas the slash-asterisk, asterisk-slash delimiters `/*` …… `*/` are used for commenting multiple lines at once.

```
/*
   Created By Bouras Aristides
   Date created: 12/25/2003
   Date modified: 04/03/2008
   Description: This program displays some messages on the screen
*/
public static void main(String[] args) throws Exception {
   System.out.println("Hello Zeus!"); //It displays a message on the screen
```

```
    //Display a second message on the screen
    System.out.println("Hello Hera!");
    /* Display a third message on screen */ System.out.println("Γεια σας");
    //This is a comment System.out.println("The End");
}
```

As you can see in the preceding program, you can add comments above a statement or at the end of it, but not in front of it. Look at the last statement, which is supposed to display the message “The End”. This statement is never executed because it is considered part of the comment.

✎ *If you add comments using the delimiters /* …… */ in front of a statement, the statement is still executed. In the preceding example, the Greek message “Γεια σας”, even though it is written next to some comments, is still executed. It is advisable, however, not to follow this writing style because it can make your code difficult to read.*

✎ *Comments are not visible to the user of a program while the program runs.*

4.16 User-Friendly Programs

What is a *user-friendly* program? It's one the user considers a friend instead of an enemy, one that is easy for a novice user.

If you want to write user-friendly programs you have to put yourself in the shoes of the user. Users want the computer to do their job their way, with a minimum of effort. Hidden menus, unclear labels and directions, and misleading error messages can all make a program user-unfriendly!

The law that best defines user-friendly designs is the *Law of Least Astonishment*: “*The program should act in a way that least astonishes the user*”. This law is also commonly referred to as the *Principle of Least Astonishment* (*POLA*).

4.17 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A recipe for a meal is actually an algorithm.
2) Algorithms are used only in computer science.
3) An algorithm can run forever.
4) In an algorithm, you can relocate a step in any position you wish.

5) An algorithm must produce the correct output values for at least one set of input values.
6) Computers can play chess.
7) An algorithm can always become a computer program.
8) Programming is the process of creating a computer program.
9) There are always three parties involved in a computer program: the programmer, the computer, and the user.
10) The programmer and the user can sometimes be the same person.
11) It is possible for a computer program to output no results.
12) A flowchart is a computer program.
13) A flowchart is composed of a set of geometric shapes.
14) A flowchart is a method used to represent an algorithm.
15) To represent an algorithm, you can design a flowchart without using any Start/End symbols.
16) You can design a flowchart without using any Process symbols.
17) You can design a flowchart without using any Data input/output symbols.
18) A flowchart must always include at least one Decision symbol.
19) In a flowchart, a Decision symbol can have one, two, or three exit paths, depending on the given problem.
20) Reserved words are all those words that have a strictly predefined meaning.
21) Structured programming includes structured design.
22) Java is a structured computer language.
23) The basic principle of structured programming is that it includes only four fundamental control structures.
24) One statement, written ten times, is considered a loop control structure.
25) Decision control structure refers to the line-by-line execution.
26) A misspelled keyword is considered a logic error.
27) A Java program can be executed even though it contains logic errors.
28) If you leave an exclamation mark at the end of a Java statement, it is considered a syntax error.

29) If you leave an exclamation mark at the end of a Java statement, it cannot prevent the whole Java program from being executed.
30) One of the advantages of structured programming is that no errors are made while writing a computer program.
31) Logic errors are caught during compilation.
32) Runtime errors are caught during compilation
33) Syntax errors are the most difficult errors to detect.
34) A program that calculates the area of a triangle but outputs the wrong results contains logic errors.
35) When a program includes no output statements, it contains syntax errors.
36) A program must always contain comments.
37) If you add comments to a program, the computer can more easily understand it.
38) You cannot add comments above a statement.
39) Comments are not visible to the users of a program.
40) A program is called user-friendly if it can be used easily by a novice user.
41) The acronym POLA stands for “Principle of Least Amusement”.

4.18 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) An algorithm is a strictly defined finite sequence of well-defined statements that provides the solution to
    a) a problem.
    b) a specific class of problems.
    c) both of the above are correct.
2) Which of the following is **not** a property that an algorithm must satisfy?
    a) effectiveness
    b) fittingness
    c) definiteness
    d) input

3) A computer program is
    a) an algorithm.
    b) a sequence of instructions.
    c) both of the above
4) When someone writes a recipe, they are the
    a) “programmer”
    b) “user”
    c) none of the above
5) Which of the following does **not** belong in the three main stages involved in creating an algorithm?
    a) data protection
    b) data input
    c) results output
    d) data processing
6) A flowchart can be
    a) presented on a piece of paper.
    b) entered directly into a computer as is.
    c) both of the above
7) A rectangle in a flowchart represents
    a) an input/output operation.
    b) a processing operation.
    c) a decision.
    d) none of the above
8) Which of the following is/are control structures?
    a) a decision
    b) a sequence
    c) a loop
    d) All of the above are control structures.
9) Which of the following Java statements contains a syntax error?
    a) `System.out.println("Hello Poseidon")`

b) `System.out.println("It's me! I contain a syntax error!!!");`

c) `System.out.println("Hello Athena");`

d) none of the above

10) Which of the following `System.out.println` statements is actually executed?

a) `System.out.println("Hello Apollo);`

b) `/* System.out.println("Hello Artemis"); */`

c) `//This will be executed// System.out.println("Hello Ares");`

d) `/* This will be executed */ System.out.println("Hello Aphrodite");`

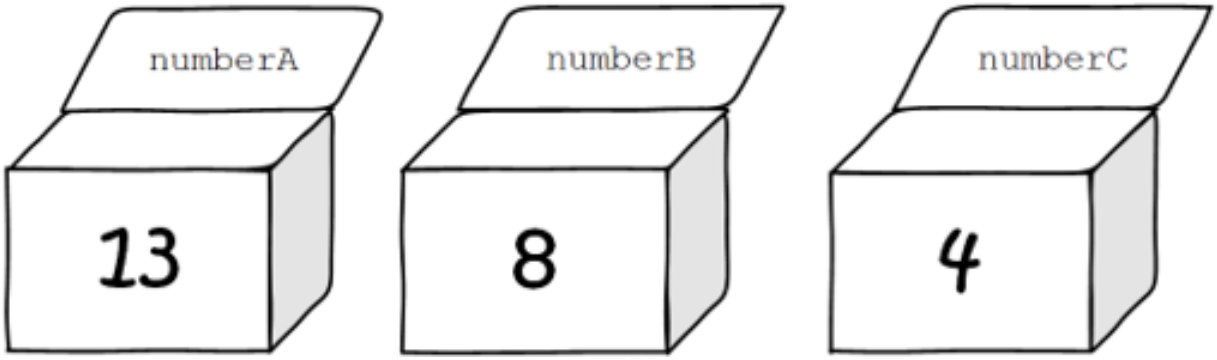
e) none of the above

Chapter 5
Variables and Constants

5.1 What is a Variable?

In computer science, a *variable* is a location in the computer's main memory (RAM) where a program can store a value and change it as the program executes.

Picture a variable as a transparent box in which you can insert and hold one thing at a time. Because the box is transparent, you can also see what it contains. Also, if you have two or more boxes you can give each box a unique name. For example, you could have three boxes, each containing a different number, and you could name the boxes `numberA`, `numberB`, and `numberC`.

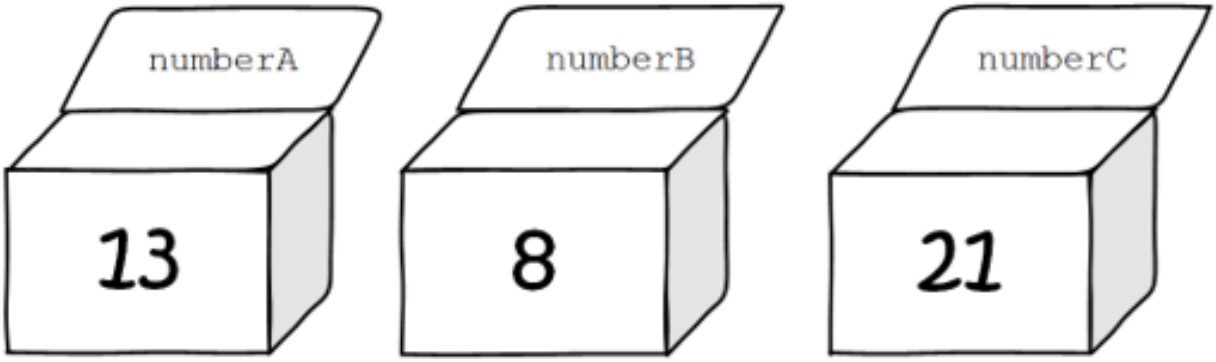


The boxes named `numberA`, `numberB` and `numberC` in the example contain the numbers 13, 8, and 4, respectively. Of course, you can examine or even alter the contained value of each one of these boxes at any time.

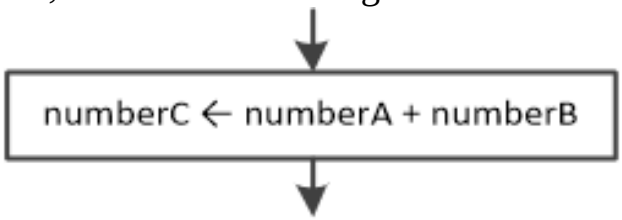
Now, let's say that someone asks you to find the sum of the values of the first two boxes and then store the result in the last box. The steps you must follow are: 1) Look at the first two boxes and examine the values they contain.

2) Use your CPU (this is your brain) to calculate the sum (the result).
3) Insert the result (which is the value of 21) in the last box. However, since each box can contain only one single value at a time, the value 4 is actually replaced by the number 21.

The boxes now look like this.



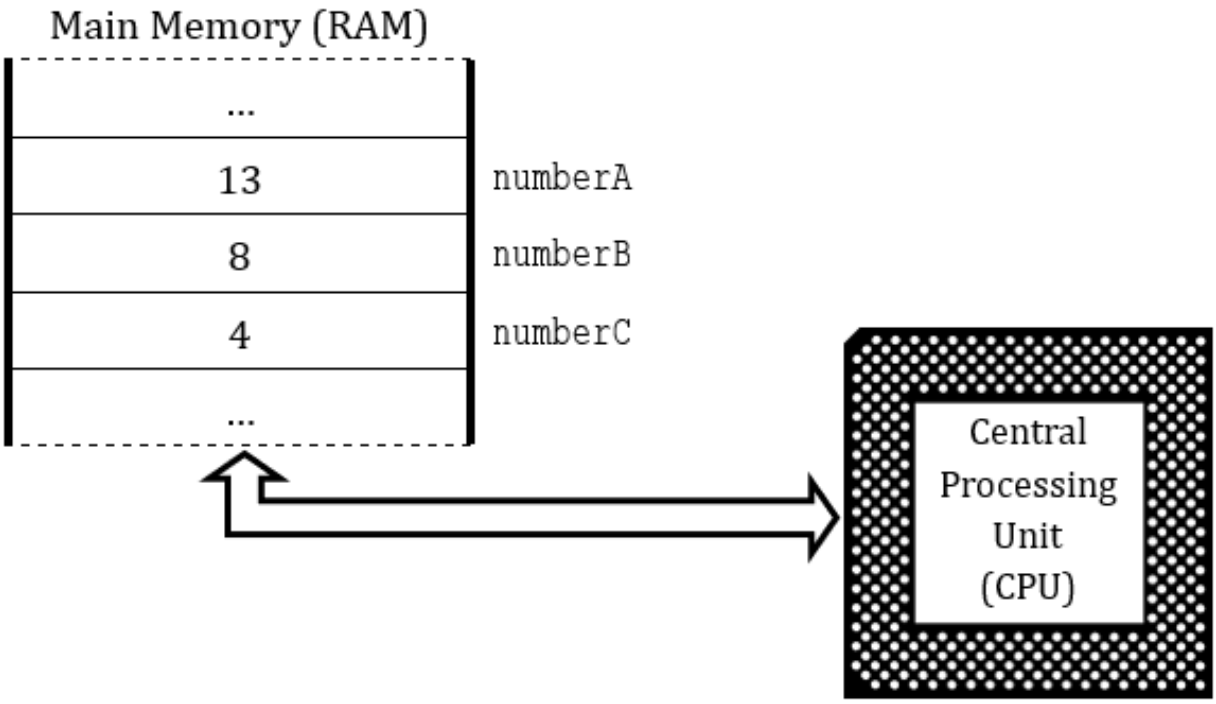
In a flowchart, the action of storing a value in a variable is represented by a left arrow



This action is usually expressed as “Assign a value, or the result of an expression, to a variable”. The left arrow is called the *value assignment operator*.

✎ *Note that this arrow always points to the left. You are not allowed to use right arrows. Also, on the left side of the arrow only one single variable must exist.*

In real computer science, the three boxes are actually three individual regions in main memory (RAM), named `numberA`, `numberB` and `numberC`.



When a program instructs the CPU to execute the following statement
`numberC` ← `numberA` + `numberB`

it follows the same three-step process as in the previous example.

1) The numbers 13 and 8 are transferred from the RAM's regions named `numberA` and `numberB` to the CPU.

   (This is the first step, in which you examined the values contained in the first two boxes).

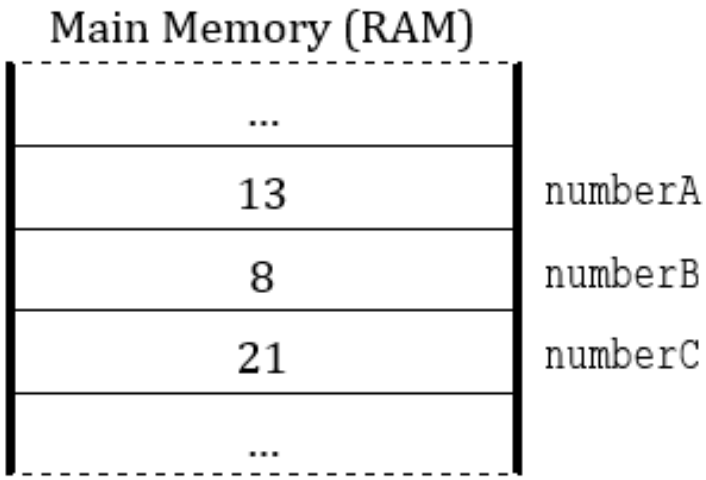
2) The CPU calculates the sum of 13 + 8.

   (This is the second step, in which you used your brain to calculate the sum, or result).

3) The result, 21, is transferred from the CPU to the RAM's region named `numberC`, replacing the existing number 4.

   (This is the third step, in which you inserted the result in the last box).

After execution, the RAM looks like this.



While a Java program is running, a variable can hold various values, but only one value at a time. When you assign a value to a variable, this value remains stored until you assign a new value replacing the old one.

The content of a variable can change to different values, but its name will always be the same because the name is just an identifier of a location in memory.

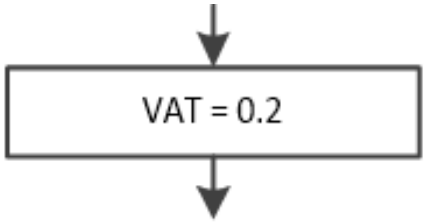
A variable is one of the most important elements in computer science because it helps you interact with data stored in the main memory (RAM). Soon, you will learn all about how to use variables in Java.

5.2 What is a Constant?

Sometimes, you may need to use a value that cannot change while the program is running. Such a value is called a *constant*. In simple terms, a constant can be thought of as a locked variable. This implies that when a program begins to run, a value is assigned to the constant, and thereafter, nothing can alter that value while the program is in progress. For example, in a financial program an interest rate can be declared as a constant.

A descriptive name for a constant can also improve the readability of your program and help you avoid some errors. For example, let's say that you are using the value 3.14159265 (but not as a constant) at many points throughout your program. If you make a typographic error when typing the number, this will produce the wrong results. But, if this value is given a name, any typographical error in the name is detected by the compiler, and you are notified with an error message.

In a flowchart, you can represent the action of setting a constant equal to a value with the equals (=) sign.



✎ *This book uses uppercase characters to distinguish a constant from a variable.*

Consider an algorithm that lets the user enter the prices of three different products and then calculates and displays the 20% Value Added Tax (known as VAT) for each product. The flowchart in **Figure 5–1** shows this process when no constant is used.

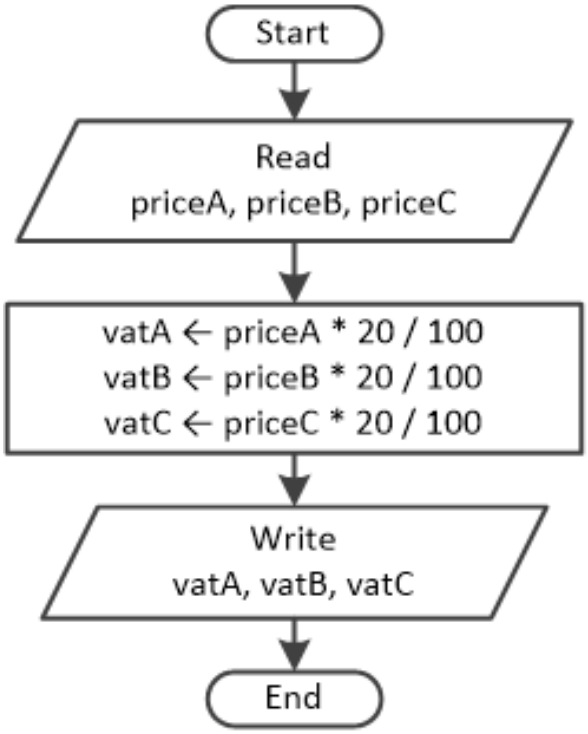


Figure 5–1 Calculating the 20% VAT for three products without the use of a constant Even though this algorithm is absolutely correct, the problem is that the author used the 20% VAT rate (20/100) three times. If this were an actual computer program, the CPU would be forced to calculate the result of the division (20/100) three individual times.

✎ *Generally speaking, division and multiplication are CPU-time consuming operations that must be avoided when possible.*

A much better solution would be to use a variable, as shown in **Figure 5–2**. This reduces the number of division operations and also decreases the potential for typographical errors.

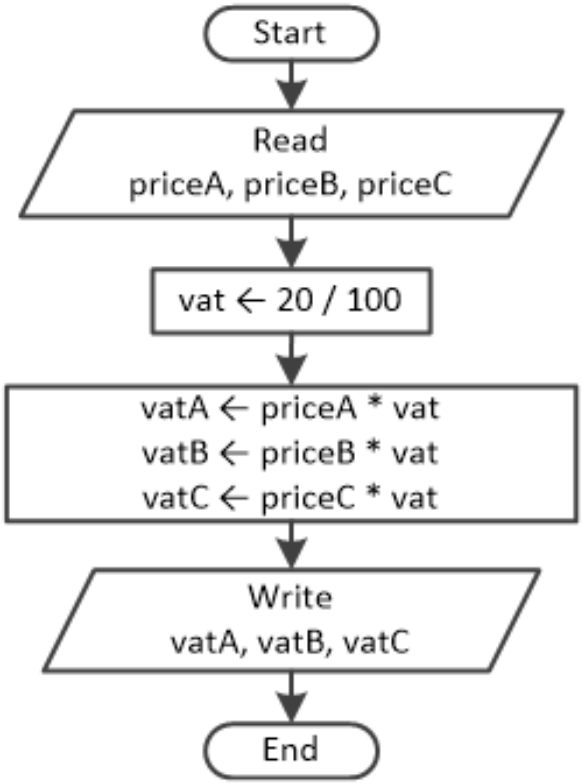


Figure 5–2 Calculating the 20% VAT for three products using a variable, `vat` This time the division (20/100) is calculated only once, and then its result is used to calculate the VAT of each product. But even now, the algorithm (which might later become a computer program) isn't perfect; `vat` is a variable and any programmer could accidentally change its value below in the program.

The ideal solution would be to change the variable `vat` to a constant `VAT`, as shown in

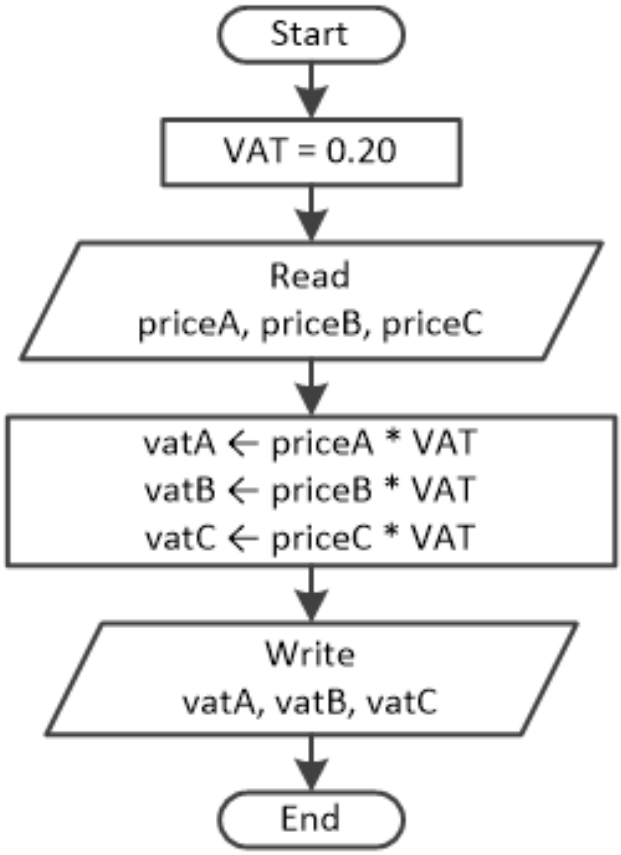


Figure 5–3 Calculating the 20% VAT for three products using a constant, `VAT`

✎ *Note that when a constant is declared in a flowchart, the equals (=) sign is used instead of the left arrow.*

This last solution is the best choice for many reasons.

- No one, including the programmer, can change the value of constant `VAT` just by accidentally writing a statement such as `VAT ← 0.60` in any position of the program.
- The potential for typographical errors is minimized.
- The number of arithmetic operations is kept as low as possible.
- If one day the finance minister decides to increase the Value Added Tax rate from 20% to 22%, the programmer will need to change just one line of code!

5.3 How Many Types of Variables and Constants Exist?

Many different types of variables and constants exist in most computer languages. The reason for this diversity is the different types of data each variable or constant can hold. Most of the time, variables and constants hold the following types of data.

- **Integers**: An *integer* value is a positive or negative number without any fractional part, such as 5, 100, 135, −25, and −5123.
- **Reals**: A *real* value is a positive or negative number that includes a fractional part, such as 5.1, 7.23, 5.0, 3.14, and −23.78976. Real values are also known as *floats*.
- **Booleans**[7]: A *Boolean* variable (or constant) can hold only one of two values: `true` or `false`.
- **Characters**: A *character* is an *alphanumeric* value (a letter, a symbol, or a digit), and it is usually enclosed in single or double quotes, such as “a”, 'c', or “@”. In computer science, a sequence of characters is known as a *string*!!! Probably the word “string” makes you visualize something wearable, but unfortunately it's not. Please keep your dirty precious mind focused on computer science! Examples of strings are “Hello Zeus”, “I am 25 years old”, or “Peter Loves Jane For Ever”.

> ✎ *In Java, strings must be enclosed in double quotes.*

5.4 Rules and Conventions for Naming Variables and Constants in Java

Certain rules must be followed when you choose a name for your variable or constant.

- The name of a variable or constant should only consist of Latin characters (English uppercase or lowercase characters), numbers, the dollar sign (`$`), and the underscore character (_). Especially for constants, even though lowercase letters are permitted, it is advisable to use only uppercase letters. This convention aids in visually distinguishing constants from variables. Examples of variable names are `firstName`, `lastName1`, and `age` while examples of constant names are `VAT`, and `COMPUTER_NAME`.
- Variable and constant names are case-sensitive, meaning there is a distinct difference between uppercase and lowercase characters. For

example, `myVAR`, `myvar`, `MYVAR`, and `MyVar` are actually four different names.

- No space characters are allowed. If a variable or constant name consists of more than one word, you can use the underscore character (_) between the words or start each word (except the first one) capitalized (Camel Case convention). For example, the variable name `student first name` is incorrect. Instead, you might use `student_first_name`, or even better, `studentFirstName`.
- A valid variable or constant name can start with a letter, or an underscore. Numbers are allowed, but they cannot be used at the beginning of the name. For example, the variable name `1studentName` is not properly written. Instead, you might use something like `studentName1` or `student1Name`.
- A variable or constant name is usually chosen in a way that describes the meaning and the role of the data it contains. For example, a variable that holds a temperature value might be named `temperature`, `temp`, or even `t`.
- Do not use any of the reserved words of Java as a variable or constant name. For example, the name `while` cannot be a valid variable or constant name since it is a reserved word in Java.

✎ *When naming variables (or constants) in Java, the dollar sign ($) character should be avoided.*

✎ *The "Camel Case convention" is a style for naming identifiers (variables, subprograms, classes etc.) in computer programming. It is called "Camel Case" because the capital letters in the middle of the name resemble the humps of a camel. There are two main variations of Camel Case: a) Lower Camel Case (or Camel Case), where the first letter of the identifier starts with a lowercase letter, and the first letter of each subsequent word starts with an uppercase letter; and b) Upper Camel Case (or Pascal Case), which is similar to Lower Camel Case, but also, the first letter of the identifier starts with an uppercase letter.*

✎ *The Lower Camel Case convention is often used for naming variables and subprograms, while the Upper Camel Case is for naming classes. You will learn more about subprograms and classes in Part VII and Part VIII correspondingly.*

5.5 What Does the Phrase “Declare a Variable” Mean?

Declaration is the process of reserving a portion in main memory (RAM) for storing the content of a variable. In many high-level computer languages (including Java), the programmer must write a specific statement to reserve that portion in the RAM before the variable can be used. In most cases, they even need to specify the variable type so that the compiler or the interpreter knows exactly how much space to reserve.

Here are some examples showing how to declare a variable in different high-level computer languages.

Declaration Statement	High-level Computer Language
Dim sum As Integer	Visual Basic
int sum;	C#, C, C++, Java, and many more
sum: Integer;	Pascal, Delphi
var sum;	Javascript

5.6 How to Declare Variables in Java

Java is a strongly typed programming language. This means that each variable must have a specific data type associated with it. In Java there are eight primitive data types: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, or `char`. Which one to use depends on the given problem! To be more specific:

- type `byte` can hold an integer between −127 and +128
- type `short` can hold an integer between −32768 and +32767
- type `int` can hold an integer between -2^{31} and $+2^{31} - 1$
- type `long` can hold an integer between -2^{63} and $+2^{63} - 1$
- type `float` can hold a real of single precision type `double` can hold a real of double precision type `boolean` can hold only two possible values: that is, `true` or `false` type `char` can hold a single character or a integer between 0 and +65535.

✎ *In many computer languages, there is one more variable type called "string", which can hold a sequence of characters. These sequences of characters, or strings are usually enclosed in double or single quotes, such as "Hello Zeus", "I am 25 years old", and so on. Java also supports strings, but keep in mind that a string in Java is not a primitive data type. Without going into detail, a string in Java is declared the same way as you declare a primitive data type but internally Java stores and handles them in a quite different way.*

To declare a variable, the general form of the Java statement is

type *name [= value]*;

where

- *type* can be `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, or even `String`.
- *name* is a valid variable name. It should follow the Lower Camel Case convention as well as all the rules for naming variables presented in Section 5.4.
- *value* is optional. If supplied, it can be any valid initial value.

Next are examples presenting the declaration of some variables in Java.

```
int number1; boolean found; String firstName; String studentName;
```

✎ *Note that, type* `String` *is written with a capital "S".*

Below are examples demonstrating the declaration and direct assignment of an initial value to some variables

int num = 5; **String** name = "*Hera*"; **char** favoriteCharacter = 'w';

which are equivalent to the following:

int num; **String** name; **char** favoriteCharacter; num = 5;
name = "*Hera*"; favoriteCharacter = 'w';

✎ *In Java, assigning a value to a variable is accomplished using the equals (=) sign. This operation is equivalent to the left arrow in flowcharts.*

✎ *Note that in Java you assign a value to a variable of type* `String` *using double quotes (" "), but you assign a value to a variable of type* `char` *using single quotes (' ').*

Last but not least, you can declare many variables of the same type on one line by separating them with commas.

```
int a, b; double x, y, z; long w = 3, u = 2;
```

5.7 How to Declare Constants in Java

You can declare constants in Java using the sequence of keywords `static` and `final`. The general form of the statement is as follows.

```
static final type NAME = value;
```

where

- *`type`* can be `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`, or even `String`.
- *`NAME`* is a valid constant name. It should follow all the rules and conventions for naming constants presented in Section 5.4.
- *`value`* is any valid value.

The following examples declare some constants in Java.

```
static final double VAT = 0.22; static final int NUMBER_OF_PLAYERS = 25; static final
String COMPUTER_NAME = "pc01"; static final String FAVORITE_SONG = "We are the world";
static final char FAVORITE_CHARACTER = 'w';
```

Note that in Java you declare a constant of type `String` *using double quotes (" "), but you declare a constant of type* `char` *using single quotes (' ').*

Once a constant is defined, its value cannot be altered while the program is running.

Java requires that all statements be terminated with a semicolon (;).

Even though the name of a constant can contain lowercase letters, it is advisable to use only uppercase letters. This helps you to visually distinguish constants from variables.

5.8 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A variable is a location in the computer's secondary storage device.
2) For a value assignment operator in a flowchart, you can use either a left or a right arrow.
3) The content of a variable can change while the program executes.
4) The content of a constant can change while the program executes.

5) The value 10.0 is an integer.
6) A Boolean variable can hold only one of two values.
7) The value “10.0” enclosed in double quotes is a real value.
8) In computer science, a string is something that you can wear!
9) The name of a variable can contain numbers.
10) A variable can change its name while the program executes.
11) The name of a variable cannot be a number.
12) The name of a constant must always be a descriptive one.
13) The name `student name` is not a valid variable name.
14) The name `STUDENT_NAME` is a valid constant name.
15) In Java, the name of a constant can contain uppercase and lowercase letters.
16) In Java, you need to declare a variable before it can be used.
17) In a Java program, you must always use at least one constant.

5.9 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) A variable is a place
   a) in a hard disk.
   b) a DVD disc.
   c) a USB flash drive.
   d) all of the above.
   e) none of the above.
2) A variable can hold
   a) one value at a time.
   b) many values at a time.
   c) all of the above.
   d) none of the above.
3) In general, using constants in a program
   a) helps programmers to completely avoid typographical errors.
   b) helps programmers to avoid using division and multiplication.
   c) all of the above.
   d) none of the above.
4) Which one of the following is an integer?
   a) 5.0
   b) −5
   c) “5”

d) none of the above is an integer.

5) A Boolean variable can hold the value
   a) one.
   b) "true".
   c) `true`.
   d) none of the above.

6) In Java, a character can be
   a) enclosed in single quotes.
   b) enclosed in double quotes.
   c) both of the above.

7) Which of the following is **not** a valid Java variable?
   a) `city_name`
   b) `cityName`
   c) `cityname`
   d) `city-name`

8) You can define a constant by using the keywords `static final`. Once a constant is defined, it
   a) can never be changed.
   b) it can be changed using the keywords `static final` again.
   c) none of the above is correct.

5.10 Review Exercises

Complete the following exercises.

1) Match each element from the first column with one element from the second column.

Value	Data Type
1. "true"	a. Boolean
2. 123	b. Real
3. false	c. String
4. 10.0	d. Integer

2) Match each element from the first column with one element from the second column.

Value	Data Type
1. The name of a person	a. Boolean
2. The age of a person	b. Real
3. The result of the division 5.0/2.0	c. Integer

4. Is it black or is it white?	d. String

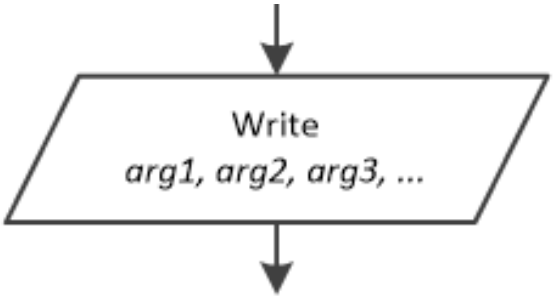
3) Complete the following table

Value	Data Type	Declaration and Initialization
The name of my friend	String	`string name = "Mark";`
My address		`string address = "254 Lookout Rd. Wilson, NY 27893";`
The average daily temperature		
A telephone number		`string phoneNumber = "1-891-764-2410";`
My Social Security Number (SSN)		
The speed of a car		
The number of children in a family		

Chapter 6
Handling Input and Output

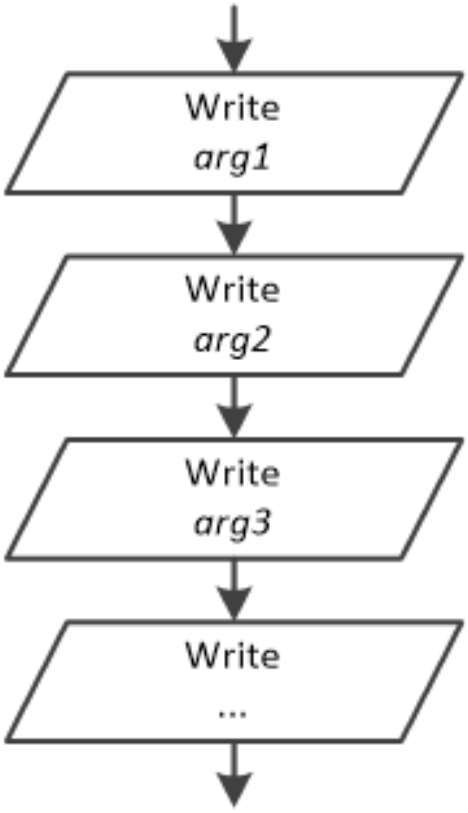
6.1 How to Output Messages and Results to a User's Screen?

A flowchart uses the oblique parallelogram and the reserved word “Write” to display a message or the final results to the user's screen.



where *arg1*, *arg2*, and *arg3* can be variables, expressions, constant values, or alphanumeric values enclosed in double quotes.

The oblique parallelogram that you have just seen is equivalent to the following flowchart fragment.



In Java, you can achieve the same result by using the `System.out.print` statement. Its general form is

System.out.**print**(*arg1* + *arg2* + *arg3* + …);

or the equivalent sequence of statements

System.out.**print**(*arg1*); System.out.**print**(*arg2*); System.out.**print**(*arg3*); …

The following code fragment:

```
a = 5;
b = 6;
c = a + b;
System.out.print("The sum of 5 and 6 is " + c);
```

display the message shown in **Figure 6–1**.

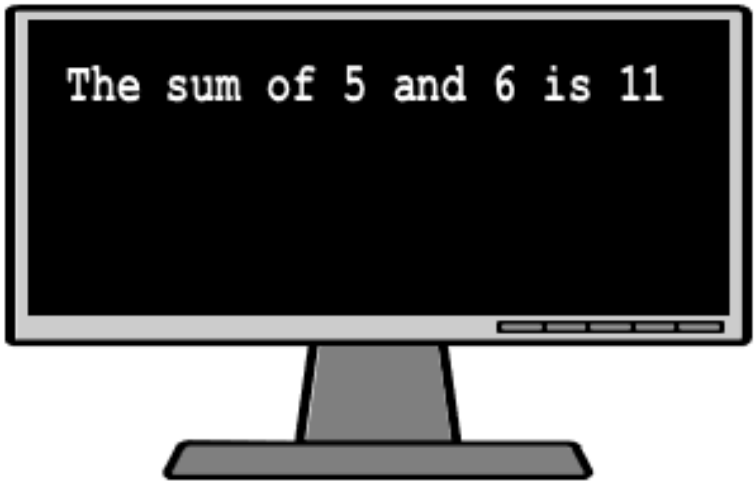


Figure 6–1 A string and an integer displayed on the screen

In Java, if you want to display a string on the screen, the string must be enclosed in double quotes.

In the last Java code fragment, note the space character at the end of the first string, just after the word “is”. If you remove it, the number 11 will get too close to the last word and the output on the screen will be `The sum of 5 and 6 is11`

The result of a mathematical expression can also be calculated directly in a `System.out.print` statement. The following code fragment displays exactly the same message as in **Figure 6–1**.

```
a = 5;
b = 6;
System.out.print("The sum of 5 and 6 is " + (a + b));
```

Note that Java requires that all statements be terminated with a semicolon.

6.2 How to Output Special Characters?

Look carefully at the following example:

```
public static void main(String[] args) throws Exception {
    System.out.print("Hello"); System.out.print("Hallo");
    System.out.print("Salut"); }
```

Although you may believe that these three messages are displayed one under the other, the actual output result is shown in **Figure 6–2**.

Figure 6–2 The output result displays on one line In order to output a "line break" you must put the special sequence of characters `\n` after every word.

```
public static void main(String[] args) throws Exception {
   System.out.print("Hello\n"); System.out.print("Hallo\n");
   System.out.print("Salut\n"); }
```

or use the Java statement `System.out.println` as follows

```
public static void main(String[] args) throws Exception {
    System.out.println("Hello"); System.out.println("Hallo");
    System.out.println("Salut"); }
```

Note that it is `println`, not `print`. The `System.out.println()` statement adds a "line break" at the end of the output.

The output result now appears in **Figure 6–3**.

Figure 6–3 The output result now displays line breaks Keep in mind that the same result can also be accomplished with one single statement.

```
System.out.println("Hello\nHallo\nSalut");
```

Another useful sequence of characters is the `\t` which can be used to create a “tab stop”. The tab character (`\t`) is useful for aligning output.

```
public static void main(String[] args) throws Exception {
   System.out.println("John\tGeorge"); System.out.println("Sofia\tMary"); }
```

The output result appears in **Figure 6–4**.

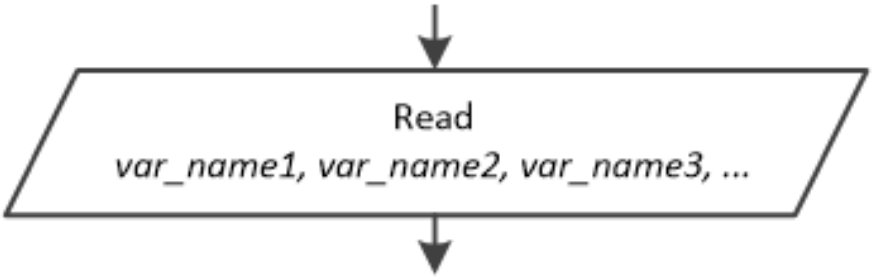
Figure 6–4 The output result displays tab characters Of course, the same result can be accomplished with one single statement.

```
System.out.println("John\tGeorge\nSofia\tMary");
```

6.3 How to Prompt the User to Enter Data?

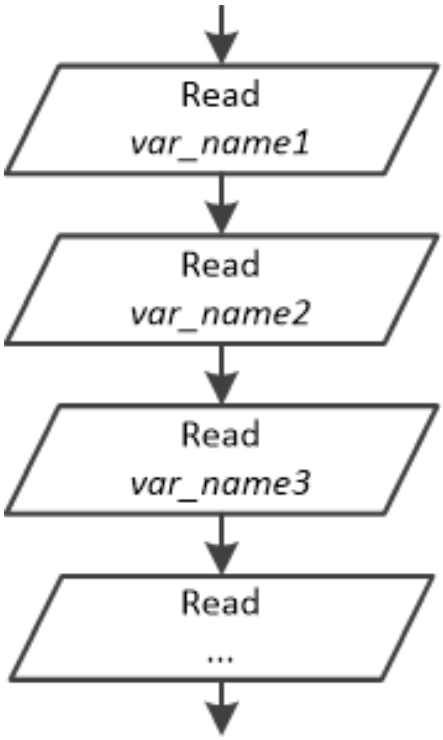
Do you recall the three main stages involved in creating an algorithm or a computer program? The first stage was the “data input” stage, in which the computer lets the user enter data such as numbers, their name, their address, or their year of birth.

A flowchart uses the oblique parallelogram and the reserved word “Read” to let a user enter their data.



where *var_name1*, *var_name2*, and *var_name3* must be variables only.

The oblique parallelogram that you have just seen is equivalent to the following flowchart fragment.



✎ *When a* `Read` *statement is executed, the flow of execution is interrupted until the user has entered all the data. When data entry is complete, the flow of execution continues to the next statement. Usually data are entered from a keyboard.*

In Java, data input is accomplished using the class `Scanner` as shown in the code that follows.

```
import java.util.Scanner;
```

```
public class App {
   static Scanner cin = new Scanner(System.in);
   public static void main(String[] args) throws Exception {
      String var_name_str;
      byte var_name_byte;
      short var_name_short;
      int var_name_int;
      long var_name_long;
      double var_name_dbl;
      //Read a string from the keyboard
      var_name_str = cin.nextLine();
      //Read a very short integer from the keyboard
      var_name_byte = Byte.parseByte(cin.nextLine());
      //Read a short integer from the keyboard
      var_name_short = Short.parseShort(cin.nextLine());
      //Read an integer from the keyboard
      var_name_int = Integer.parseInt(cin.nextLine());
      //Read a long integer from the keyboard
      var_name_long = Long.parseLong(cin.nextLine());
      //Read a real from the keyboard
      var_name_dbl = Double.parseDouble(cin.nextLine());
   }
}
```

where

- *var_name_str* can be any variable of type `String`.
- *var_name_byte* can be any variable of type `byte`.
- *var_name_short* can be any variable of type `short`.
- *var_name_int* can be any variable of type `int`.
- *var_name_long* can be any variable of type `long`.
- *var_name_dbl* can be any variable of type `double`.

✎ *In order to use the class* `Scanner` *you need to import the library* `Scanner` *into your project using the statement* `import java.util.Scanner`.

✎ *The statement* `static Scanner cin = new Scanner(System.in)` *creates the object* `cin` *of the class* `Scanner`. *You will learn more about classes and objects in Part VIII. Just be patient!*

The following example lets the user enter their name and then displays it with the word “Hello” in front of it.

```
String name;
name = cin.nextLine(); System.out.print("Hello " + name);
```

When the `cin.nextLine()` statement of this example executes, the flow of execution stops, waiting for the user to enter their name. The `System.out.print("Hello " + name)` statement is not yet executed! As long as the user doesn't enter anything, the computer just waits, as shown in **Figure 6–5**.

Figure 6–5 When a `cin.nextLine()` statement executes, the computer waits for data input.

When the user finally enters their name and hits the "Enter ↵" key, the flow of execution then continues to the next `System.out.print()` statement as shown in **Figure 6–6.**

Figure 6–6 The flow of execution continues when the user hits the "Enter ↵" key.

However, the previous example could be improved if, before each data input, a "prompt" message is displayed. This makes the program more

user-friendly. For example, look at the following code fragment.

```
String name;
System.out.print("What is your name? "); name = cin.nextLine();
System.out.print("Hello " + name);
```

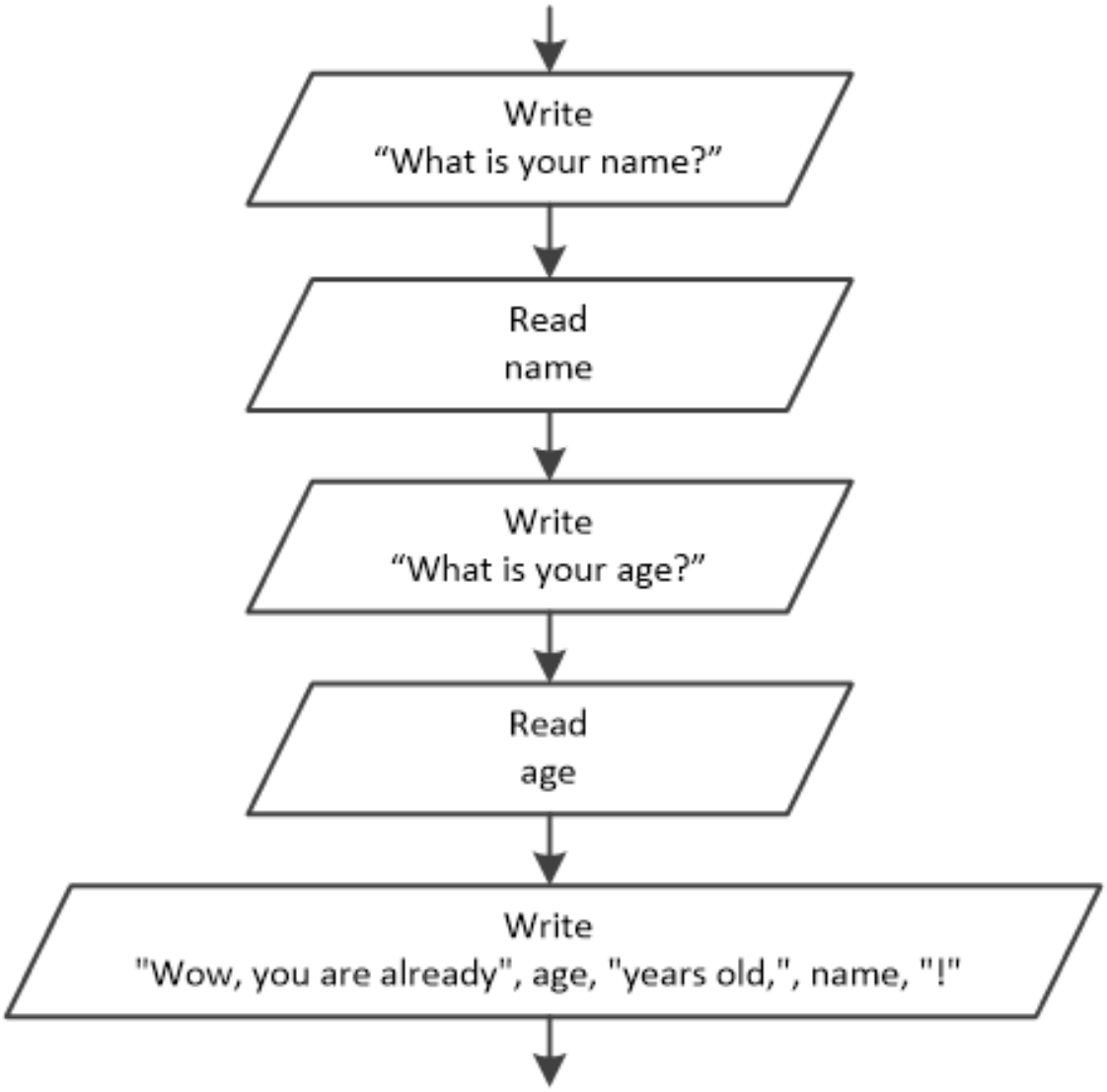
In the preceding code fragment, before the `cin.nextLine()` statement executes, the message “What is your name?” (without the double quotes) is displayed, as shown in **Figure 6–7**.

Figure 6–7 When a “prompt” message is displayed before the `cin.nextLine()` statement.

The following code fragment prompts the user to enter their name and age.

```
String name; byte age;
System.out.print("What is your name? "); name = cin.nextLine();
System.out.print("What is your age? "); age = Byte.parseByte(cin.nextLine());
System.out.print("Wow, you are already " + age + " years old, " + name + "!");
```

The corresponding flowchart fragment looks like this.



To read a float (a double), that is, a number that contains a fractional part, you need to use a slightly different statement. The following code fragment prompts the user to enter the name and the price of a product.

```
String productName; double productPrice;
System.out.print("Enter product name: "); productName = cin.nextLine();
System.out.print("Enter product price: "); productPrice =
Double.parseDouble(cin.nextLine());
```

In this book there is a slight difference between the words “prompts” and “lets”. When an exercise says “Write a Java program that **prompts** the user to enter…” this means that you *must* include a prompt message. However, when the exercise says “Write a Java program that **lets** the user enter…” this means that you are not actually required to include a prompt message; that is, it is not wrong to include one but you don't have to! The

following example lets the user enter their name and age (but does not prompt them to).

```
name = cin.nextLine(); age = Byte.parseByte(cin.nextLine());
```

What happens here (when the program is executed) is that the computer displays a text cursor without any prompt message and waits for the user to enter two values—one for `name` and one for `age`. The user, though, must be a prophet and guess what to enter! Do they have to enter their name first and then their age, or is it the opposite? So, obviously a prompt message is pretty much required, because it makes your program more user-friendly.

6.4 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) In Java, the word `System` is a reserved word.
2) The `System.out.print()` statement can be used to display a message or the content of a variable.
3) When the `cin.nextLine()` statement is executed, the flow of execution is interrupted until the user has entered a value.
4) One single `cin.nextLine()` statement can be used to enter multiple data values.
5) Before data input, a prompt message must always be displayed.

6.5 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The statement `System.out.print(hello)` displays
   a) the word “hello” (without the double quotes).
   b) the word “hello” (including the double quotes).
   c) the content of the variable `hello`.
   d) none of the above
2) The statement `System.out.print("HELLO")` displays
   a) the word “HELLO” (without the double quotes).
   b) the word “HELLO” (including the double quotes).
   c) the content of the constant `HELLO`.

d) none of the above

3) The statement `System.out.print("Hello\nHermes")` displays

a) the message "Hello Hermes" (without the double quotes).

b) the word "Hello" in one line and the word "Hermes" in the next one (without the double quotes).

c) the message "HelloHermes" (without the double quotes).

d) the message "Hello\nHermes" (without the double quotes).

e) none of the above

4) The statement `data1data2 = cin.nextLine()`

a) lets the user enter a value and assigns it to variable `data1`. Variable `data2` remains empty.

b) lets the user enter a value and assigns it to variable `data1data2`.

c) lets the user enter two values and assigns them to variables `data1` and `data2`.

d) none of the above

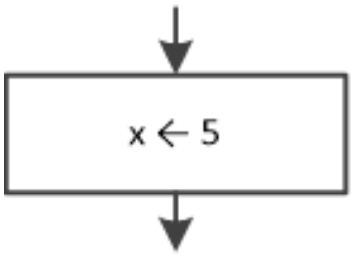
Chapter 7
Operators

7.1 The Value Assignment Operator

The most commonly used operator in Java is the value assignment operator (=). For example, the following Java statement assigns a value of 5 to variable x.

```
x = 5;
```

As you read in Chapter 5, this is equivalent to the left arrow used in flowcharts.



Probably the left arrow used in a flowchart is more convenient and clearer than the (=) sign because it visually illustrates that the value or the result of an expression on the right is assigned to a variable on the left.

It's important to note that the (=) sign is not equivalent to the one used in mathematics. In mathematics, the expression `x = 5` is read as "`x` *is equal to* `5`". However, in Java the expression `x = 5` is read as "*assign the value* `5` *to* `x`" or "*set* `x` *equal to* `5`". They look the same but they act differently!

For instance, in mathematics, the following two lines are equivalent. The first one can be read as "`x` *is equal to the sum of* `y` *and* `z`" and the second one as "*the sum of* `y` *and* `z` *is equal to* `x`".

```
x = y + z
y + z = x
```

On the other hand, in Java, these two statements are definitely **not** equivalent.

```
x = y + z;
y + z = x;
```

The first statement is a valid Java statement, conveying "*Assign the sum of* `y` *and* `z` *to* `x`". The second statement, however, is invalid, as it attempts to assign the value of `x` to `y + z` , which is not permissible in Java!

In Java, the variable on the left side of the (=) sign represents a region in main memory (RAM) where a value can be stored.

On the left side of the (=) sign only one single variable must exist, whereas on the right side there can be a number, a variable, a string, or even a complex mathematical expression.

In **Table 7-1** you can find some examples of value assignments.

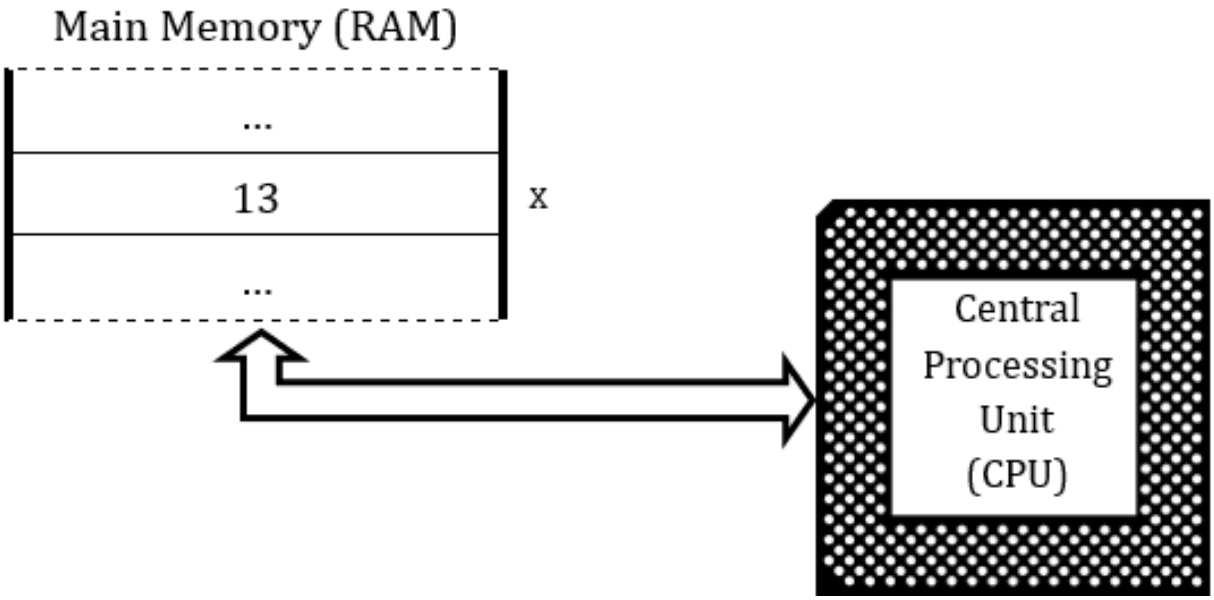
Examples	Description
`a = 9;`	Assign a value of 9 to variable `a`.
`b = c;`	Assign the content of variable `c` to variable `b`.
`d = "Hello Zeus";`	Assign the string *Hello Zeus* to variable `d`.
`d = a + b;`	Calculate the sum of the contents of variables `a` and `b` and assign the result to variable `d`.
`b = x + 1;`	Calculate the sum of the content of variable `x` and 1 and assign the result to variable `b`. Please note that the content of variable `x` is not altered.
`x = x + 1;`	Calculate the sum of the content of variable `x` and 1 and assign the result back to variable `x`. In other words, increase variable `x` by one.

Table 7-1 Examples of value assignments Confused about the last one? Are you thinking about your math teachers right now? What would their reaction be if you had written `x = x + 1` on the blackboard? Can you personally imagine a number that equals itself plus one? This statement suggests that 5 is equal to 6 and 10 is equal to 11, which is, of course, incorrect!

Obviously, things are different in computer science. The statement `x = x + 1` is absolutely valid! It instructs the CPU to retrieve the value of variable `x` from main memory (RAM), to add 1 to that value, and to assign the result back to variable `x`. The old value of variable `x` is replaced by the new one.

Still don't get it? Let's take a look at how the CPU and main memory (RAM) cooperate with each other in order to execute the statement `x = x + 1`.

Let's say that there is a region in memory, named `x` and it contains the number 13.

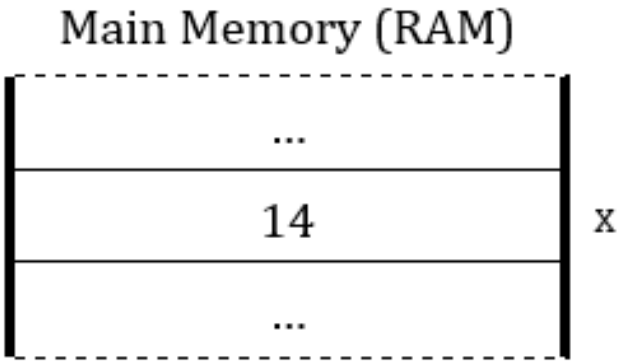


When a program instructs the CPU to execute the statement:

```
x = x + 1;
```

the following procedure is carried out: ► the number 13 is transferred from the RAM's region named `x` to the CPU; ► the CPU calculates the sum of 13 and 1; and ► the result, 14, is transferred from the CPU to the RAM's region `x` replacing the existing number, 13.

After execution, the RAM looks like this.



Now that you have understood everything, let's delve into one last detail. In Java, you can assign a single value to multiple variables with one single statement. The following statement assigns the value of 4 to all three variables `a`, `b`, and `c`.

```
a = b = c = 4;
```

7.2 Arithmetic Operators

Just like every high-level programming language, Java supports almost all types of *arithmetic operators*.

Arithmetic Operator	Description
+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division (Modulus)

The first two operators are straightforward and need no further explanation.

If you need to multiply two numbers or the content of two variables you have to use the asterisk (*) symbol. For example, if you want to multiply 2 times y, you must write `2 * y`.

> ✎ *In mathematics it is legal to skip the multiplication operator and write* `3x`*, meaning "3 times x". In Java, however, you must always use an asterisk anywhere a multiplication operation exists. This is one of the most common mistakes novice programmers make when they write mathematical expressions in Java.*

To perform a division, you must use the slash (/) symbol. For example, if you want to divide 10 by 2, you must write `10 / 2`.

However, it's important to note that in Java, the result of the division of two integers is always an integer. Thus, in the expression `7 / 2`, since both numbers 7 and 2 are integers, the results is 3 (rather than 3.5, as one might mistakenly expect). In contrast, in the expression `7.0 / 2`, where at least one of the numbers is a real (float), the result is indeed 3.5.

The following three statements are equivalent. They all output the value of 3.5.

```
System.out.println(7.0 / 2); System.out.println(7 / 2.0); System.out.println(7.0 / 2.0);
```

The modulus operator (%) returns the remainder of an integer division, which means that the statement

```
c = 13 % 3;
```

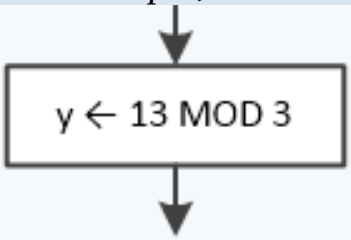
assigns a value of 1 to variable `c`.

The modulus operator (%) can be used with floating-point numbers as well, but the result is a real (float). For example, the operation

```
d = 14.4 % 3;
```

assigns a value of 2.4 (and not 2, as you may mistakenly expect) to variable `d`.

✎ *Keep in mind that flowcharts are a loose method used to represent an algorithm. Although the use of the modulus (%) operator is allowed in flowcharts, this book uses the commonly accepted MOD operator instead! For example, the Java statement* `y = 13 % 3` *is represented in a flowchart as*



In mathematics, as you may already know, you are allowed to use parentheses (round brackets) as well as braces (curly brackets) and square brackets, as presented in the following expression.

$$y = \frac{5}{2}\left\{3 + 2\left[4 + 7\left(6 - \frac{4}{3}\right)\right]\right\}$$

However, in Java there is no such thing as braces and brackets. Parentheses are all you have; therefore, the same expression must be written using parentheses instead of braces or brackets.

```
y = 5.0 / 2.0 * (3 + 2 * (4 + 7 * (6 – 4.0 / 3.0)));
```

7.3 What is the Precedence of Arithmetic Operators?

Arithmetic operators follow the same precedence rules as in mathematics, and these are: multiplication and division are performed first, addition and subtraction are performed afterwards.

Higher Precedence ⇧ Lower precedence	Arithmetic Operators
	*, /, %
	+, −

When multiplication and division exist in the same expression, and since both are of the same precedence, they are performed left to right (the same way as you read), which means that the expression

```
y = 6 / 3 * 2;
```

is equivalent to $y = \frac{6}{3} \cdot 2$, and assigns a value of 4 to variable `y`, (division is performed before multiplication).

If you want, however, the multiplication to be performed before the division, you can use parentheses to change the precedence. This means that

```
y = 6 / (3 * 2);
```

is equivalent to $y = \frac{6}{3 \cdot 2}$, and assigns a value of 1 to variable `y` (multiplication is performed before division).

> ✎ *Keep in mind that it is not possible in Java to write fractions in the form of* $\frac{6}{3}$ *or* $\frac{4x+5}{6}$*. Forget it! There is no equation editor in Visual Studio Code, or in any IDE. All fractions must be written on one single line. For example,* $\frac{6}{3}$ *must be written as* `6 / 3`*, and* $\frac{4x+5}{6}$ *must be written as* `(4 * x + 5) / 6`*.*

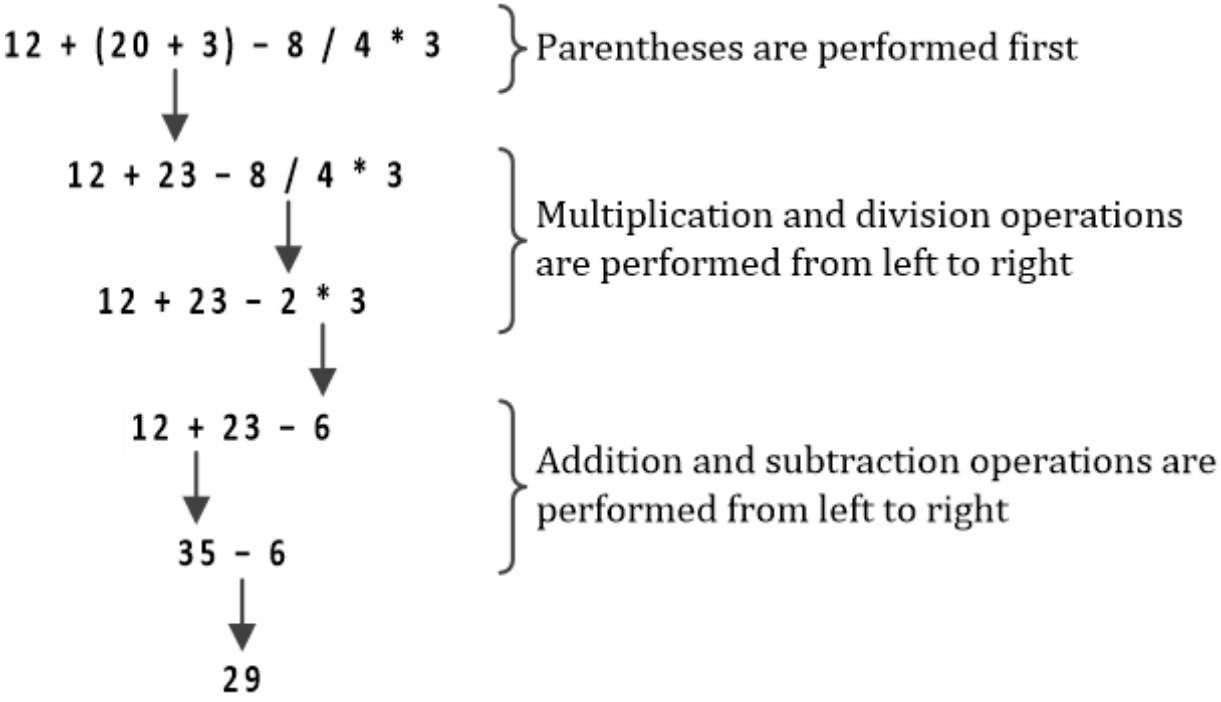
The order of operations can be summarized as follows: 1) Any operations enclosed in parentheses are performed first.

2) Next, any multiplication and division operations are performed from left to right.
3) In the end, any addition and subtraction operations are performed from left to right.

So, in the next example

```
y = 12 + (20 + 3) - 8 / 4 * 3;
```

the operations are performed as follows:


12 + (20 + 3) - 8 / 4 * 3
Parentheses are performed first
12 + 23 - 8 / 4 * 3
12 + 23 - 2 * 3
Multiplication and division operations are performed from left to right
12 + 23 - 6
35 - 6
29
Addition and subtraction operations are performed from left to right

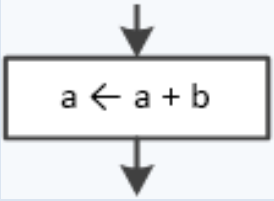
7.4 Compound Assignment Operators

Java offers a special set of operators known as *compound assignment operators*, which can help you write code faster. These operators are comprehensively detailed in the table below. An example for each operator is provided, while the *"Equivalent to"* column shows the corresponding statement without using a compound assignment operator.

Operator	Description	Example	Equivalent to
+=	Addition assignment	`a += b;`	`a = a + b;`
−=	Subtraction assignment	`a −= b;`	`a = a − b;`
*=	Multiplication assignment	`a *= b;`	`a = a * b;`
/=	Division assignment	`a /= b;`	`a = a / b;`
%=	Modulus assignment	`a %= b;`	`a = a % b;`

✎ *Bear in mind that in flowcharts, this book only uses the commonly accepted operators shown in the "Equivalent to" column. For example, the*

Java statement `a += b` *is represented in a flowchart as*



Exercise 7.4-1 Which Java Statements are Syntactically Correct?

Which of the following Java assignment statements are syntactically correct?

i) `a = -10;`

ii) `10 = b;`

iii) `aB = aB + 1;`

iv) `a = "COWS";`

v) `a = COWS;`

vi) `a + b = 40;`

vii) `a = 3 b;`

viii) `a = "true";`

ix) `a = true;`

x) `a /= 2;`

xi) `a += 1;`

xii) `a =* 2;`

Solution i) Correct. It assigns the integer value −10 to variable a.

ii) **Wrong**. On the left side of the value assignment operator, only variables can exist.

iii) **Correct**. It increases variable `aB` by one.

iv) **Correct**. It assigns the string “COWS” (without the double quotes) to variable `a`.

v) **Correct**. It assigns the content of constant (or even variable) `COWS` to variable `a`.

vi) **Wrong**. On the left side of the value assignment operator, only variables (not expressions) can exist.

vii) **Wrong**. It should have been written as `a = 3 * b`.

viii) **Correct**. It assigns the string “true” (without the double quotes) to variable `a`.

ix) **Correct**. It assigns the value `true` to variable `a`.

x) **Correct**. This is equivalent to `a = a / 2`.

xi) **Correct**. This is equivalent to `a = a + 1`.

xii) **Wrong**. It should have been be written as `a *= 2` (which is equivalent to `a = a * 2`).

Exercise 7.4-2 Finding Variable Types

What is the type of each of the following variables?

i) `a = 15;`

ii) `width = "10 meters";`

iii) `b = "15";`

iv) `temp = 13.5;`

v) `b = true;`

vi) `b = "true";`

Solution i) The value 15 belongs to the set of integers, thus the variable `a` is an integer.

ii) The value “*10 meters*” is a text, thus the `width` variable is a string.

iii) The value “15” is a text, thus the `b` variable is a string.

iv) The value 13.5 belongs to the set of real numbers, thus the variable `temp` is real (float).

v) The value `true` is Boolean, thus the variable `b` is a Boolean.

vi) The value “true” is a text, thus the variable `b` is a string.

7.5 Incrementing/Decrementing Operators

Both, adding 1 to a number, or subtracting 1 from a number, are so frequently used operations in computer programming that Java incorporates a special set of operators to do this. Java supports two types of incrementing and decrementing operators: ► pre-incrementing/decrementing operators ► post-incrementing/decrementing operators Pre-incrementing/decrementing operators are placed before the variable name, while post-incrementing /decrementing operators are placed after the variable name. These four types of operators are shown here.

Operator	Description	Example	Equivalent to
Pre-incrementing	Increment a variable by one	`++a;`	`a = a + 1;`
Pre-decrementing	Decrement a variable by one	`--a;`	`a = a - 1;`
Post-incrementing	Increment a variable by one	`a++;`	`a = a + 1;`
Post-decrementing	Decrement a variable by one	`a--;`	`a = a - 1;`

As you can see in the *"Equivalent to"* column, it is obvious that you can achieve the same result by simply using the classic assignment operator (=). Nevertheless, opting for these new operators is not only more productive but also enhances efficiency.

Let's see an example with incrementing operators,

```
a = b = 5;
++a; //This is equivalent to a = a + 1
b++; //This is equivalent to b = b + 1
System.out.println(a); //It displays: 6
System.out.println(b); //It displays: 6
```

The double slashes (`//`) *indicate that the text that follows is a comment; thus, it is never executed.*

In the previous example, the pre- and post-incrementing operators increment the variables `a` and `b` by one! So, where is the catch? Are these two operators equivalent? The answer is "yes", but only in this specific example. In other cases the answer will likely be "no". There is a small difference between the two operators.

Let's spot that difference! The rule is that a pre-incrementing/decrementing operator performs the increment/decrement operation first and then delivers the new value. A post-incrementing/decrementing operator delivers the old value first and then performs the increment/decrement operation. Look carefully at the next two examples.

```
a = 5;
b = ++a;
System.out.println(a); //It displays: 6
System.out.println(b); //It displays: 6
```

and

```
a = 5;
```

```
b = a++;
System.out.println(a); //It displays: 6
System.out.println(b); //It displays: 5
```

In the first example, variable a is incremented by one and then its new value is assigned to variable b. In the end, both variables contain a value of 6.

In the second example, the value 5 of variable a is assigned to variable b, and then variable a is incremented by one. In the end, variable a contains a value of 6 but variable b contains a value of 5!

To enhance efficiency in a program, incrementing/decrementing operators can be used directly in an expression, as demonstrated in the following example.

```
a1 = a2 = 5;
b = --a1 * 2 - 1;
System.out.println(b); //It displays: 7
System.out.println(a2++ * 2); //It displays: 10
System.out.println(a2); //It displays: 6
```

7.6 String Operators

Joining two separate strings into a single one is called *concatenation*. There are two operators that you can use to concatenate (join) strings as shown in the table that follows.

Operator	Description	Example	Equivalent to
+	Concatenation	`a = "Hi" + " there";`	
+=	Concatenation assignment	`a += "Hello";`	`a = a + "Hello";`

The following code fragment displays “What's up, dude?”

```
String a, b, c;
a = "What's "; b = "up, "; c = a + b;
c += "dude?";
System.out.println(c);
```

Exercise 7.6-1 Concatenating Names

Write a Java program that prompts the user to enter their first and last name (assigned to two different variables). It then joins them in a single string (concatenation) and displays them on the user's screen.

Solution The Java program is shown here.

```
public static void main(String[] args) throws Exception {
   String firstName, lastName, fullName;
   System.out.print("Enter first name: "); firstName = cin.nextLine();
   System.out.print("Enter last name: "); lastName = cin.nextLine();
   fullName = firstName + " " + lastName; System.out.println(fullName); }
```

✎ *Note the extra space character added between the first and last name.*

7.7 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) The statement `x = 5` can be read as *"Variable x is equal to 5"*.
2) The value assignment operator assigns the result of an expression to a variable.
3) A string can be assigned to a variable only by using the `cin.nextLine()` statement.
4) The statement `5 = y` assigns value 5 to variable `y`.
5) On the right side of a value assignment operator an arithmetic operator must always exist.
6) On the right side of a value assignment operator only variables can exist.
7) `You cannot use the same variable on both sides of a value assignment operator.`
8) The statement `a = a + 1` decrements variable `a` by one.
9) The statement `a = a + (-1)` decrements variable `a` by one.
10) In Java, the word `MOD` is a reserved word.
11) The statement `x = 0 % 5` assigns a value of 5 to variable `x`.
12) The operation `5 % 0` is not possible.
13) Addition and subtraction have the higher precedence among the arithmetic operators.
14) When division and multiplication operators co-exist in an expression, multiplication operations are performed before division.
15) The expression `8 / 4 * 2` is equal to 1.
16) The expression `4 + 6 / 6 + 4` is equal to 9.

17) The expression `a + b + c / 3.0` calculates the average value of three numbers.
18) The statement `a += 1` is equivalent to `a = a + 1`
19) The statement `a = "true"` assigns a Boolean value to variable `a`.
20) The statement `a = 2·a` doubles the content of variable `a`.
21) The statements `a += 2` and `a = a − (−2)` are not equivalent.
22) The statement `a −= a + 1` always assigns a value of `−1` to variable `a`.
23) The statement `a = "George" + " Malkovich"` assigns the value "GeorgeMalkovich" (without the double quotes) to the variable `a`.
24) The following code fragment satisfies the property of definiteness.

```
double a, b, x;
a = Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine()); x
= a / (b - 7);
System.out.println(x);
```

7.8 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) Which of the following Java statements assigns a value of 10.0 to variable `a`?
   a) `10.0 = a;` b) `a ← 10.0;` c) `a = 100.0 / 10.0;` d) none of the above
2) The statement `a = b` can be read as
   a) assign the content of variable `a` to variable `b`.
   b) variable `b` is equal to variable `a`.
   c) assign the content of variable `b` to variable `a`.
   d) none of the above
3) The expression `0 % 10 + 2` is equal to
   a) 7.
   b) 2.
   c) 12.
   d) none of the above
4) Which of the following Java statements is syntactically correct?
   a) `a = 4 * 2y − 8 / (4 * d);` b) `a = 4 * 2 * y − 8 / 4 * d;` c) `a = 4 * 2 * y − 8 / (4 */ d);` d) none of the above
5) Which of the following Java statements is syntactically correct?
   a) `a * 5 = a;` b) `a = a * 5;` c) `a =* 5;` d) none of the above
6) Which of the following Java statements assigns the value "George Malkovich"

(without the double quotes) to the variable `a`?

a) `a = "George" + " " + "Malkovich";` b) `a = "George" + "Malkovich";` c) `a = "George " + "Malkovich";` d) all of the above

7) The following code fragment

```
x = 2;
x++;
```

does **not** satisfy the property of a) finiteness.

b) definiteness.

c) effectiveness.

d) none of the above

8) The following code fragment

```
double a, x; a = Double.parseDouble(cin.nextLine()); x = 1 / a;
```

does **not** satisfy the property of a) finiteness.

b) input.

c) definiteness.

d) none of the above

7.9 Review Exercises

Complete the following exercises.

1) Which of the following Java assignment statements are syntactically correct?

i) `a ← a + 1;` ii) `a += 1;` iii) `a b = a b + 1;` iv) `a = a + 1` v) `a = hello;` vi) `a = 40;` vii) `a = b + 1;` viii) `a =+ "true";` ix) `fdadstwsdgfgw = x;` x) `a = a * 5;`

2) What is the type of each of the following variables?

i) `a = "false";` ii) `w = false;` iii) `b = "15 meters";` iv) `weight = "40";` v) `b = 13.0;` vi) `b = 13;`

3) Match each element from the first column with one element from the second column.

Operation	Result
i) 1 / 2.0	a) 100
ii) 1.0 / 2 * 2	b) 0.25
iii) 0 % 10 * 10	c) 0
iv) 10 % 2 + 7	d) 0.5
	e) 7

	f) 1.0

4) What displays on the screen after executing each of the following code fragments?

i)
```
a = 5;
b = a * a + 1;
System.out.println(b++);
```

ii)
```
a = 9;
b = a / 3 * a;
System.out.println(b + 1);
```

5) What displays on the screen after executing each of the following code fragments?

i)
```
a = 5;
a += a − 5;
System.out.println(a);
```

ii)
```
a = 5;
a = a + 1;
System.out.println(a);
```

6) What is the result of each of the following operations?

i) `21 % 5`

ii) `10 % 2`

iii) `11 % 2`

iv) `10 % 6 % 3`

v) `0 % 3`

vi) `100 / 10 % 3`

7) What displays on screen after executing each of the following code fragments?

i)
```
a = 5;
b = 2;
c = a % (b + 1);
```

d = (b + 1) % (a + b);
System.out.**println**(c + " * " + d);

ii)

a = 4;
b = 8;
a += 1;
double c = a * b / 10 % b; System.out.**println**(c);

8) Calculate the result of the expression `a % b` for the following cases.

i) `a = 20, b = 3`

ii) `a = 15, b = 3`

iii) `a = 22, b = 3`

iv) `a = 0, b = 3`

v) `a = 3, b = 1`

vi) `a = 2, b = 2`

9) Calculate the result of the expression `b * (a % b) + a / b` for each of the following cases.

i) `a = 10, b = 5`

ii) `a = 10, b = 2`

10) What displays on the screen after executing the following code fragment?

```
a = "My name is"; a += " ";
a = a + "George Malkovich"; System.out.println(a);
```

11) Fill in the gaps in each of the following code fragments so that they both display a value of 5.

i)

a = 2;
a = a - …… ;
System.out.**println**(a);

ii)

a = 4;
b = a * 0.5;
b += a;
a = b - …… ;
System.out.**println**(a);

12) What displays on the screen after executing the following code fragment?

```
city = "California"; California = city;
System.out.println(city + " " + California + " California");
```

Chapter 8
Trace Tables

8.1 What is a Trace Table?

A *trace table* is a technique used to test algorithms or computer programs for logic errors that occur while the algorithm or program executes.

The trace table simulates the flow of execution. Statements are executed step by step, and the values of variables change as an assignment statement is executed.

Trace tables are useful for educational purposes. They are typically employed by novice programmers to help them visualize how a particular algorithm or program works and to assist them in detecting logic errors.

A typical trace table is shown here.

Step	Statement	Notes	variable1	variable2	variable3
1					
2					
…					

Let's see a trace table in action! For the following Java program, a trace table is created to determine the values of the variables in each step.

```
public static void main(String[] args) throws Exception {
   int x, y, z;
   x = 10; y = 15; z = x * y; z++;
   System.out.println(z); }
```

The trace table for this program is shown below. Notes are optional, but they help the reader to better understand what is really happening.

Step	Statement	Notes	x	y	z
1	`x = 10`	The value 10 is assigned to variable `x`.	**10**	?	?
2	`y = 15`	The value 15 is assigned to variable `y`.	10	**15**	?
3	`z = x * y`	The result of the product `x * y` is assigned to `z`.	10	15	**150**
4	`z++`	Variable `z` is incremented by one.	10	15	**151**

5	`.println(z)`	It displays: 151

Exercise 8.1-1 Creating a Trace Table

Create a trace table to determine the values of the variables in each step of the Java program for two different executions.

The input values for the two executions are: (i) 0.3, and (ii) 4.5.

```
public static void main(String[] args) throws Exception {
   double a, b, c;
   b = Double.parseDouble(cin.nextLine()); c = 3; c = c * b; a = 10 * c; a = a % 10;
   System.out.println(a); }
```

Solution i) For the input value of 0.3, the trace table looks like this.

Step	Statement	Notes	a	b	c
1	`b = Double.parseD…`	User enters value 0.3	?	**0.3**	?
2	`c = 3`		?	0.3	**3.0**
3	`c = c * b`		?	0.3	**0.9**
4	`a = 10 * c`		**9.0**	0.3	0.9
5	`a = a % 10`		**9.0**	0.3	0.9
6	`.println(a)`	It displays: 9.0			

ii) For the input value of 4.5, the trace table looks like this.

Step	Statement	Notes	a	b	c
1	`b = Double.parseD…`	User enters value 4.5	?	**4.5**	?
2	`c = 3`		?	4.5	**3.0**
3	`c = c * b`		?	4.5	**13.5**
4	`a = 10 * c`		**135.0**	4.5	13.5
5	`a = a % 10`		**5.0**	4.5	13.5
6	`.println(a)`	It displays: 5.0			

Exercise 8.1-2 Creating a Trace Table

What result is displayed when the following program is executed?

```
public static void main(String[] args) throws Exception {
   String Ugly, Beautiful, Handsome;
```

```
Ugly = "Beautiful"; Beautiful = "Ugly"; Handsome = Ugly;
System.out.println("Beautiful"); System.out.println(Ugly);
System.out.println(Handsome); }
```

Solution Let's create a trace table to find the output result.

Step	Statement	Notes	Ugly	Beautiful	Handsome
1	`Ugly = "Beautiful"`	The string “Beautiful” is assigned to the variable `Ugly`.	**Beautiful**	?	?
2	`Beautiful = "Ugly"`	The string “Ugly” is assigned to the variable `Beautiful`.	Beautiful	**Ugly**	?
3	`Handsome = Ugly`	The value of variable `Ugly` is assigned to the variable `Handsome`.	Beautiful	Ugly	**Beautiful**
4	`.println("Beautiful")`	It displays: Beautiful			
5	`.println(Ugly)`	It displays: Beautiful			
6	`.println(Handsome)`	It displays: Beautiful			

Exercise 8.1-3 Swapping Values of Variables

Write a Java program that lets the user enter two values, in variables `a` *and* `b`*. At the end of the program, the two variables must swap their values. For example, if variables* `a` *and* `b` *contain the values 5 and 7 respectively, after swapping their values, variable* `a` *must contain the value 7 and variable* `b` *must contain the value 5!*

Solution The following code fragment, even though it may seem correct, is erroneous and doesn't really swap the values of variables `a` and `b`!

```
int a, b;
a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
a = b;
b = a;
System.out.println(a); System.out.println(b);
```

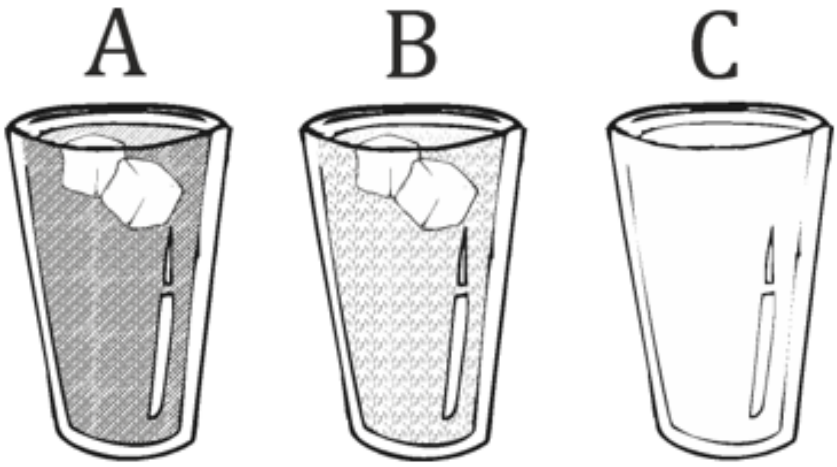
Let's see why! Suppose the user enters two values, 5 and 7. The trace table is shown here.

Step	Statement	Notes	a	b
1	`a = Integer.parseI…`	User enters the value 5	**5**	?
2	`b = Integer.parseI…`	User enters the value 7	5	**7**
3	`a = b`	The value of variable `b` is assigned to variable `a`. Value 5 is lost!	**7**	7
4	`b = a`	The value of variable `a` is assigned to variable `b`	7	**7**
5	`.println(a)`	It displays: 7		
6	`.println(b)`	It displays: 7		

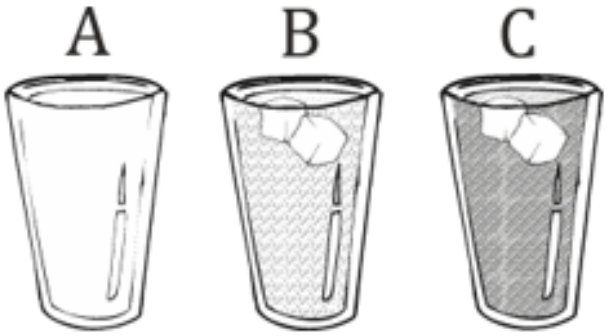
Oops! Where is the value 5?

The solution wasn't so obvious after all! So, how do you really swap values anyway?

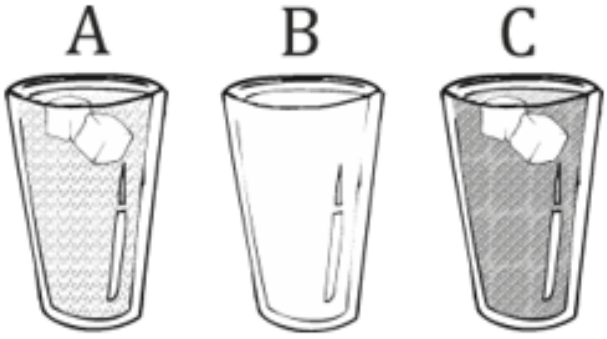
Consider two glasses: a glass of orange juice (called glass A), and a glass of lemon juice (called glass B). If you want to swap their content, all you must do is find and use one extra empty glass (called glass C).



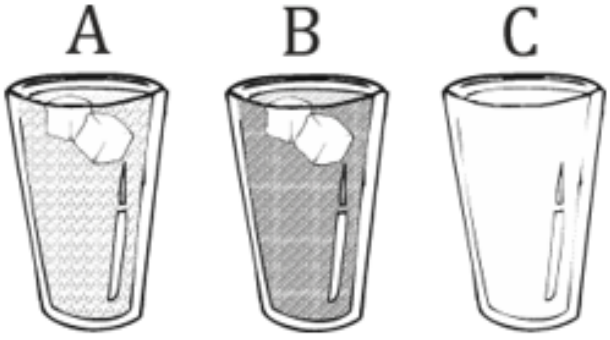
The steps that must be followed are: 1) Empty the contents of glass A (orange juice) into glass C.



2) Empty the contents of glass B (lemon juice) into glass A.



3) Empty the contents of glass C (orange juice) into glass B.



Swapping completed successfully!

You can follow the same steps to swap the contents of two variables in Java.

```
public static void main(String[] args) throws Exception {
   int a, b, c;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   c = a; //Empty the contents of glass A (orange juice) into glass C
   a = b; //Empty the contents of glass B (lemon juice) into glass A b = c; //Empty the
   contents of glass C (orange juice) into glass B
   System.out.println(a); System.out.println(b); }
```

The text after double slashes (`//`) is considered a comment and is never executed.

Exercise 8.1-4 Swapping Values of Variables – An Alternative Approach

Write a Java program that lets the user enter two integer values, in variables `a` *and* `b`*. In the end, the two variables must swap their values. Then, use a trace table with input values 5 and 7 to confirm the correctness of your code.*

Solution Since the variables contain numeric values, you can use the following Java program (as an alternative approach).

```
public static void main(String[] args) throws Exception {
   int a, b;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   a = a + b; b = a - b; a = a - b;
   System.out.println(a + " " + b); }
```

Let's now use a trace table with input values 5 and 7 to confirm that the variables `a` and `b` correctly swap their content.

Step	Statement	Notes	a	b
1	`a = Integer.parseI…`	User enters value 5	**5**	?
2	`b = Integer.parseI…`	User enters value 7	5	**7**
3	`a = a + b`		**12**	7
4	`b = a - b`		12	**5**
5	`a = a - b`		**7**	5
6	`.println(a + " " + b)`	It displays: 7 5		

The disadvantage of this method is that it cannot swap the contents of alphanumeric variables (strings).

8.2 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A trace table is a technique for testing a computer.
2) Trace tables help a programmer find errors in a computer program.
3) You cannot execute a computer program without first creating its corresponding trace table.
4) In order to swap the values of two integer variables, you always need an extra variable.

8.3 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step of the Java program when a value of 3 is entered.

```
public static void main(String[] args) throws Exception {
   double a, b, c, d;
   a = Double.parseDouble(cin.nextLine());
   b = a + 10; a = b * (a - 3); c = 3 * b / 6; d = c * c; d--;
   System.out.println(d); }
```

2) Create a trace table to determine the values of the variables in each step of the Java program for three different executions.

   The input values for the three executions are: (i) 3, (ii) 4, and (iii) 1.

```
public static void main(String[] args) throws Exception {
   int a, b, c, d;
   a = Integer.parseInt(cin.nextLine());
   a = (a + 1) * (a + 1) + 6 / 3 * 2 + 20; b = a % 13; c = b % 7; d = a * b * c;
   System.out.println(a + ", " + b + ", " + c + ", " + d); }
```

3) Create a trace table to determine the values of the variables in each step of the Java program for two different executions.

   The input values for the two executions are: (i) 8, 4; and (ii) 4, 4

```
public static void main(String[] args) throws Exception {
   int a, b, c, d, e;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   c = a + b; d = 1 + a / b * c + 2; e = c + d; c += d + e; e--;
   d -= c + d % c; System.out.println(c + ", " + d + ", " + e); }
```

Chapter 9
Using Visual Studio Code

9.1 Write, Execute and Debug Java Programs

So far, you have learned some solid basics about Java programming. Now it's time to explore the process of entering programs into the computer, executing them, observing their performance, examining how they display results, and learning techniques for debugging them.

Debugging is the process of finding and reducing the number of defects (bugs) in a computer program to make it perform as expected.

As stated in Section 2.6, an Integrated Development Environment (IDE) is a type of software that enables programmers to write, execute and debug their source code. Visual Studio Code is such an example.

All the instructions you need regarding how to write, execute and debug Java programs on either Windows or Linux are maintained on my website at the following addresses. This gives me the flexibility to review them frequently and keep them up-to-date.

https://tinyurl.com/mvsa5mzn

https://www.bouraspage.com/java-setup-write-execute-debug

If you find any inconsistencies, please let me know, and I will update the instructions as soon as possible. To report issues, visit one of the following addresses:

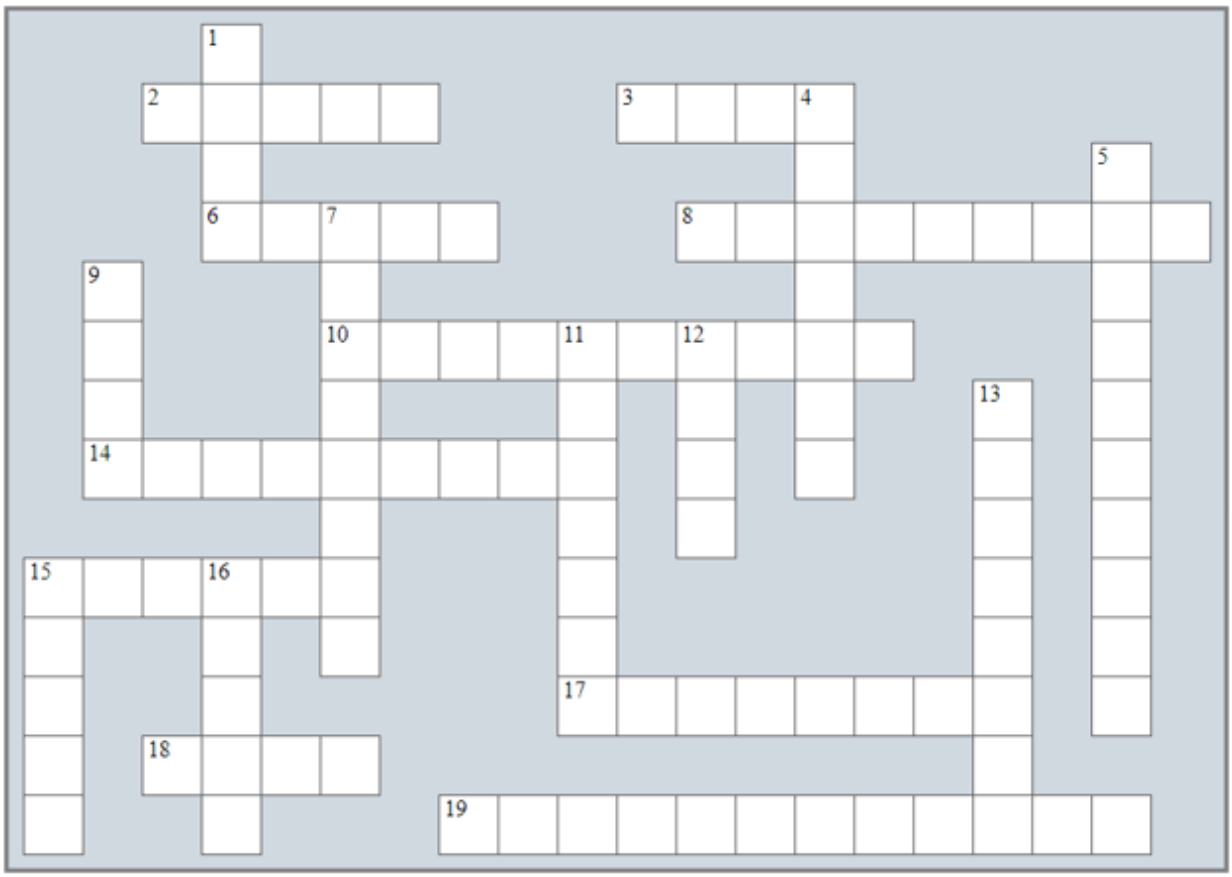
https://tinyurl.com/28nwh2nf

https://www.bouraspage.com/report-errata

Review in “Getting Started with Java”

Review Crossword Puzzles

1) Solve the following crossword puzzle.



Across

2) These errors are hard to detect.

3) A control structure.

6) It shows the flow of execution in a flowchart.

8) A graphical method of presenting an algorithm.

10) _____ programming is a software development method that uses modularization and structured design.

14) Strictly defined finite sequence of well-defined statements that provides the solution to a problem.

15) The term _____ means that the algorithm must reach an end point and cannot run forever.

17) This flowchart symbol has one entrance and two exits.

18) Logic errors and runtime errors are commonly referred to as _____.

19) It must be possible to perform each step of the algorithm correctly and in a finite amount of time. This is one of the properties an algorithm must satisfy, and it is known as _____.

Down

1) The principle that best defines user-friendly designs.

4) This represents a mathematical (formula) calculation in a flowchart.

5) Data _____ is one of the three main stages involved in creating an algorithm.

7) A word that has a strictly predefined meaning in a computer language.

9) A programming language.

11) Statement.

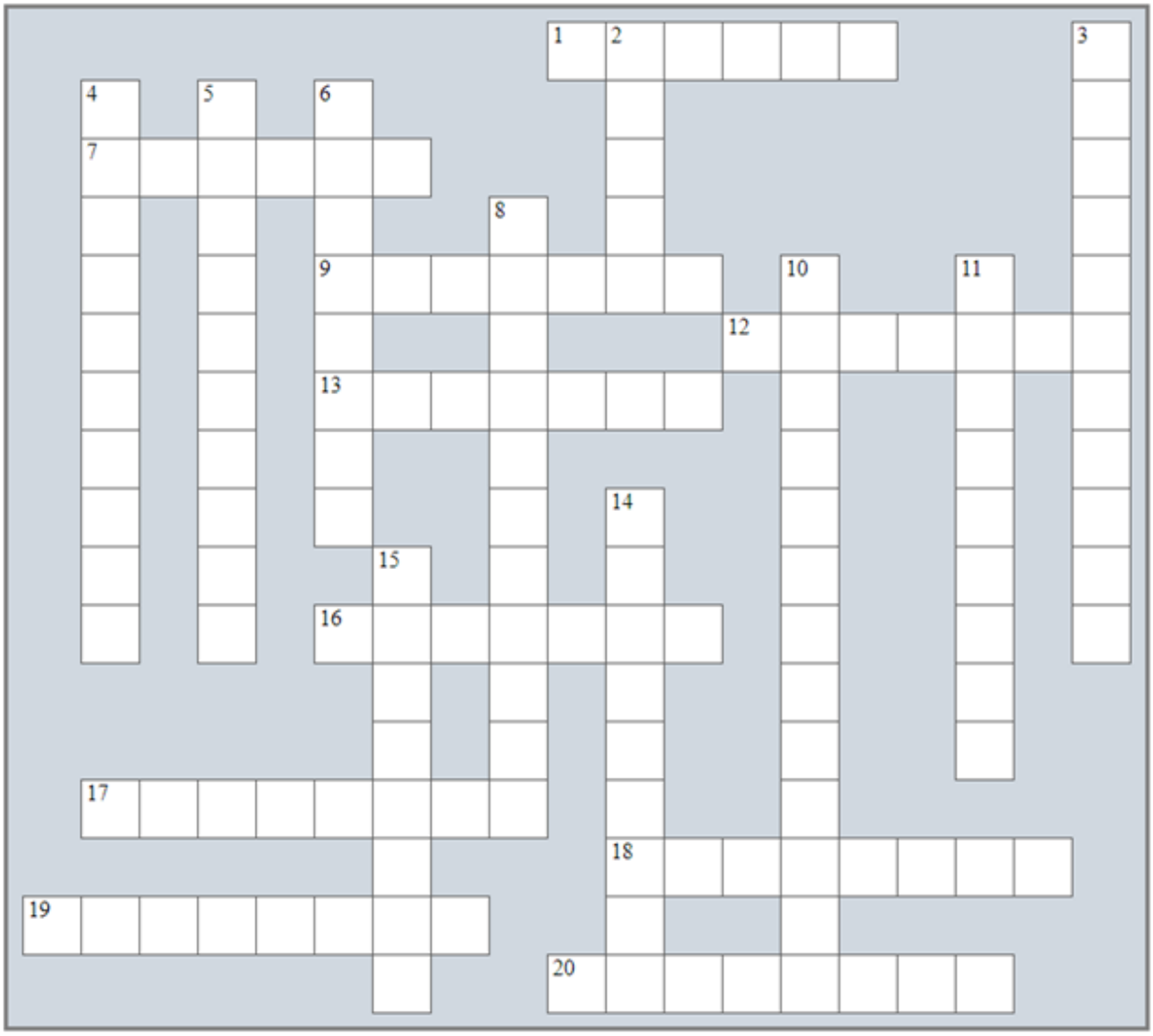
12) The person who uses a program.

13) A control structure.

15) Real.

16) One of the properties an algorithm must satisfy.

2) Solve the following crossword puzzle.



Across

1) An alphanumeric value.

7) A misspelled keyword is a _____ error.

9) A positive or negative number without any fractional part.

12) Extra information that can be included in a program to make it easier to read and understand.

13) This type of variable can hold only one of two values.

16) An error that occurs during the execution of a program.

17) This control structure is also known as a selection control structure.

18) This is a CPU-time consuming arithmetic operation.

19) A value that cannot change while the program is running.

20) A user-_____ program is one that is easy for a novice user.

Down

2) A _____ table is a technique used to test algorithms or computer programs for logic errors that occur while the algorithm or program executes.

3) Any arithmetic operations enclosed in _____ are performed first.

4) The left arrow in flowcharts is called the value _____ operator.

5) A symbol character permitted in a variable name.

6) It represents a location in the computer's main memory (RAM) where a program can store a value.

8) The process of reserving a portion in main memory (RAM) for storing the contents of a variable.

10) Joining two separate strings into a single one.

11) The process of finding and reducing the number of logic errors in a computer program.

14) The modulus operator returns the _____ of an integer division.

15) The operator (/) returns the _____ of a division.

Review Questions

Answer the following questions.

1) What is an algorithm?
2) Give the algorithm for making a cup of coffee.
3) What are the five properties an algorithm must satisfy?
4) Can an algorithm execute forever?
5) What is a computer program?
6) What are the three parties involved in an algorithm?
7) What are the three stages that make up a computer program?
8) Can a computer program be made up of two stages?

9) What is a flowchart?
10) What are the basic symbols that flowcharts use?
11) What is meant by the term “reserved words”?
12) What is structured programming?
13) What are the three fundamental control structures of structured programming?
14) Give an example of each control structure using flowcharts.
15) Can a programmer write Java programs in a text editor?
16) What is a syntax error? Give one example.
17) What is a logic error? Give one example.
18) What is a runtime error? Give one example.
19) What type of error is caused by a misspelled keyword?
20) What does the term “debugging” mean?
21) Why should programmers add comments in their code?
22) Why should programmers write user-friendly programs?
23) What does the acronym POLA stand for?
24) What is a variable?
25) How many variables can exist on the left side of the left arrow in flowcharts?
26) In which part of a computer are the values of the variables stored?
27) What is a constant?
28) How can constants be used to help programmers?
29) Why should a programmer avoid division and multiplication operations whenever possible?
30) Name at least three primitive data types of variables in Java.
31) What does the phrase “declare a variable” mean?
32) How do you declare a variable in Java? Give an example.
33) How do you declare a constant in Java? Give an example.
34) What symbol is used in flowcharts to display a message?
35) What are the special character sequences for a “line break” and “tab stop” in Java?

36) Which symbol is used in flowcharts to let the user enter data?
37) Which character is used in Java as a value assignment operator, and how is it represented in a flowchart?
38) Which arithmetic operators does Java support?
39) What is a modulus operator?
40) Summarize the rules for the precedence of arithmetic operators.
41) What compound assignment operators does Java support?
42) What incrementing/decrementing operators does Java support?
43) What string operators does Java support?
44) What is a trace table?
45) What are the benefits of using a trace table?
46) Describe the steps involved in swapping the contents (either numeric or alphanumeric) of two variables.
47) Two methods for swapping the values of two variables have been proposed in this book. Which one is better, and why?
48) Describe the way in which Visual Studio Code helps you find syntax errors.
49) Describe the ways in which Visual Studio Code helps you find logic errors.

Part III
Sequence Control Structures

Chapter 10
Introduction to Sequence Control Structures

10.1 What is the Sequence Control Structure?

Sequence control structure refers to the line-by-line execution by which statements are executed sequentially, in the same order in which they appear in the program, without skipping any of them. They might, for example, carry out a series of read or write operations, arithmetic operations, or assignments to variables.

The following program shows an example of Java statements that are executed sequentially.

Project_10.1

```
public static void main(String[] args) throws Exception {
   double num, result;
   //Prompt the user to enter value for num System.out.print("Enter a number: "); num
   = Double.parseDouble(cin.nextLine());
   //Calculate the square of num result = num * num;
   //Display the result on user's screen System.out.println("The square of " + num +
   " is " + result); }
```

The sequence control structure is the simplest of the three fundamental control structures that you learned about in Section 4.11. The other two structures are "decision structure" and "loop structure". All problems in computer programming can be solved using only these three structures!

In Java, you can add comments using double slashes (//). Comments are for human readers. Compilers and interpreters ignore them.

Exercise 10.1-1 Calculating the Area of a Rectangle

Write a Java program that prompts the user to enter the length of the base and the height of a rectangle, and then calculates and displays its area.

Solution You probably know from school that you can calculate the area of a rectangle using the following formula: Area = Base × Height In Section 4.6, you learned about the three main stages involved in creating an algorithm: data input, data processing, and results output.

In this exercise, these three main stages are as follows: ► **Data input** – the user must enter values for *Base* and *Height* ► **Data processing** – the program must calculate the area of the rectangle ► **Results output** – the program must display the area of the rectangle calculated in the previous stage.

The solution to this problem is shown here.

Project_10.1-1

```
public static void main(String[] args) throws Exception {
   double area, b, h;
   //Data input - Prompt the user to enter values for base and height
   System.out.print("Enter the length of base: "); b =
   Double.parseDouble(cin.nextLine()); System.out.print("Enter the length of height:
   "); h = Double.parseDouble(cin.nextLine());
   //Data processing - Calculate the area of the rectangle area = b * h;
   //Results output - Display the result on user's screen System.out.println("The
   area of the rectangle is " + area); }
```

Exercise 10.1-2 Calculating the Area of a Circle

Write a Java program that calculates and displays the area of a circle.

Solution You can calculate the area of a circle using the following formula: Area = π · Radius²

The value of π is a known quantity, which is approximately 3.14159. Therefore, the only value the user must enter is the value for *Radius*.

In this exercise, the three main stages that you learned in Section 4.6 are as follows: ► **Data input** – the user must enter a value for *Radius* ► **Data processing** – the program must calculate the area of the circle ► **Results output** – the program must display the area of the circle calculated in the previous stage.

The solution to this problem is shown here.

Project_10.1-2a

```
public static void main(String[] args) throws Exception
                                {
                   double area, radius;
    //Data input - Prompt the user to enter a value for
    radius System.out.print("Enter the length of radius:
      "); radius = Double.parseDouble(cin.nextLine());
```

```
    //Data processing - Calculate the area of the circle
            area = 3.14159 * radius * radius;
      //Results output - Display the result on user's
    screen System.out.println("The area of the circle is
                        " + area); }
```

A much better approach would be with to use a constant, `PI`.

```
                Project_10.1-2b
          static final double PI = 3.14159;
public static void main(String[] args) throws Exception
                        {
                double area, radius;
     //Data input - Prompt the user to enter a value for
    radius System.out.print("Enter the length of radius:
      "); radius = Double.parseDouble(cin.nextLine());
    //Data processing - Calculate the area of the circle
                area = PI * radius * radius;
      //Results output - Display the result on user's
    screen System.out.println("The area of the circle is
                        " + area); }
```

✎ *Note that the constant* `PI` *is declared outside of the method* `main`.

Exercise 10.1-3 Where is the Car? Calculating Distance Traveled

A car starts from rest and moves with a constant acceleration along a straight horizontal road for a specified time. Write a Java program that prompts the user to enter the acceleration and the time the car traveled, and then calculates and displays the distance traveled. The required formula is $S = u_o + \frac{1}{2}at^2$

*where ► **S** is the distance the car traveled, in meters (m) ► **u_o** is the initial velocity (speed) of the car, in meters per second (m/sec) ► **t** is the time the car traveled, in seconds (sec) ► **a** is the acceleration, in meters per second^2 (m/sec^2) Solution Since the car starts from rest, the initial velocity (speed) u_0 is zero. Thus, the formula becomes* $S = \frac{1}{2}at^2$

and the Java program is Project_10.1-3

```
public static void main(String[] args) throws Exception {
   double S, a, t;
   System.out.print("Enter acceleration: "); a = Double.parseDouble(cin.nextLine());
   System.out.print("Enter time traveled: "); t = Double.parseDouble(cin.nextLine());
   S = 0.5 * a * t * t; System.out.println("Your car traveled " + S + " meters"); }
```

Exercise 10.1-4 Kelvin to Fahrenheit

Write a Java program that converts a temperature value from degrees Fahrenheit[8] to its degrees Kelvin[9] equivalent. The required formula is 1.8 × Kelvin = Fahrenheit + 459.67

Solution The formula given cannot be used in your program as is. In a computer language such as Java, it is not permitted to write

1.8 * kelvin = fahrenheit + 459.67;

> *In the position on the left side of the (=) sign, only a variable must exist. This variable is actually a region in RAM where a value can be stored.*

According to the wording of this exercise, the program must convert degrees Fahrenheit to degrees Kelvin. The value for degrees Fahrenheit is a known value and it is provided by the user, whereas the value for degrees Kelvin is what the Java program must calculate. So, you need to solve for Kelvin. After a bit of work, the formula becomes

$$Kelvin = \frac{Fahrenheit + 459.67}{1.8}$$

and the Java program is shown here.

Project_10.1-4

```
public static void main(String[] args) throws Exception {
   double fahrenheit, kelvin;
   System.out.print("Enter a temperature in Fahrenheit: "); fahrenheit =
   Double.parseDouble(cin.nextLine());
   kelvin = (fahrenheit + 459.67) / 1.8;
   System.out.println("The temperature in Kelvin is " + kelvin); }
```

Exercise 10.1-5 Calculating Sales Tax

An employee needs a program to enter the before-tax price of a product and calculate its final price. Assume a value added tax (VAT) rate of 19%.

Solution The sales tax can be easily calculated. You must multiply the before-tax price of the product by the VAT rate. Be careful—the sales tax is not the final price, but only the tax amount.

The after-tax price can be calculated by adding the initial before-tax price and the sales tax that you calculated beforehand.

In this program you can use a constant named `VAT` for the sales tax rate (VAT rate).

Project_10.1-5

```
static final double VAT = 0.19;
public static void main(String[] args) throws Exception {
   double priceAfterTax, priceBeforeTax, salesTax;
   System.out.print("Enter the before-tax price: "); priceBeforeTax =
   Double.parseDouble(cin.nextLine());
   salesTax = priceBeforeTax * VAT; priceAfterTax = priceBeforeTax + salesTax;
   System.out.println("The after-tax price is: " + priceAfterTax); }
```

Note that the constant `VAT` is declared outside of the method `main`.

Exercise 10.1-6 Calculating a Sales Discount

Write a Java program that prompts the user to enter the price of an item and the discount rate offered (on a scale of 0 to 100). The program must then calculate and display the new price.

Solution The discount amount can be easily calculated. You must multiply the before-discount price of the product by the discount value and then divide it by 100. The division is necessary since the user enters a value for the discount on a scale of 0 to 100. Be careful—the result is not the final price but only the discount amount.

The final after-discount price can be calculated by subtracting the discount amount that you calculated beforehand from the initial before-discount price.

Project_10.1-6

```
public static void main(String[] args) throws Exception {
   int discount; double discountAmount, priceAfterDiscount, priceBeforeDiscount;
   System.out.print("Enter the price of a product: "); priceBeforeDiscount =
   Double.parseDouble(cin.nextLine());
```

```
   System.out.print("Enter the discount offered (0 - 100): "); discount =
   Integer.parseInt(cin.nextLine());
   discountAmount = priceBeforeDiscount * discount / 100; priceAfterDiscount =
   priceBeforeDiscount - discountAmount;
   System.out.println("The price after discount is: " + priceAfterDiscount); }
```

Exercise 10.1-7 Calculating a Sales Discount and Tax

Write a Java program that prompts the user to enter the before-tax price of an item and the discount rate offered (on a scale of 0 to 100). The program must then calculate and display the new price. Assume a sales tax rate of 19%.

Solution This exercise is just a combination of the previous two exercises!

Project_10.1-7

```
static final double VAT = 0.19;
public static void main(String[] args) throws Exception {
   int discount; double discountAmount, priceAfterDiscount, priceAfterTax; double
   priceBeforeDiscount, salesTax;
   System.out.print("Enter the price of a product: "); priceBeforeDiscount =
   Double.parseDouble(cin.nextLine());
   System.out.print("Enter the discount offered (0 - 100): "); discount =
   Integer.parseInt(cin.nextLine());
   discountAmount = priceBeforeDiscount * discount / 100; priceAfterDiscount =
   priceBeforeDiscount - discountAmount;
   salesTax = priceAfterDiscount * VAT; priceAfterTax = priceAfterDiscount +
   salesTax;
   System.out.println("The discounted after-tax price is: " + priceAfterTax); }
```

10.2 Review Exercises

Complete the following exercises.

1) In the United States, a car's fuel economy is measured in miles per gallon, or MPG. A car's MPG can be calculated using the following formula: $MPG = \frac{miles\ driven}{gallons\ of\ gas\ used}$

   Write a Java program that prompts the user to enter the total number of miles they have driven and the gallons of gas used. Then the program must calculate and display the car's MPG.

2) Write a Java program that prompts the user to enter values for base and height, and then calculates and displays the area of a

triangle. The required formula is

$$Area = \frac{1}{2} Base \times Height$$

3) Write a Java program that prompts the user to enter two angles of a triangle, and then calculates and displays the third angle.

   Hint: The sum of the measures of the interior angles of any triangle is 180 degrees.

4) Write a Java program that lets a student enter their grades from four tests, and then calculates and displays the average grade.

5) Write a Java program that prompts the user to enter a value for radius, and then calculates and displays the perimeter of a circle. The required formula is *Perimeter = 2πR*

6) Write a Java program that prompts the user to enter a value for diameter in meters, and then calculates and displays the volume of a sphere. The required formula is

$$V = \frac{4}{3} \pi R^3$$

   where *R* is the radius of the sphere.

7) Regarding the previous exercise, which of the following results output statements are correct? Which one would you choose to display the volume of the sphere on the user's screen, and why?

   a) `System.out.println(V);`
   b) `System.out.println(V cubic meters);`
   c) `System.out.println(V + cubic meters);`
   d) `System.out.println("The volume of the sphere is: " V);`
   e) `System.out.println("The volume of the sphere is: " + V);`
   f) `System.out.println("The volume of the sphere is: " + V + cubic meters);`
   g) `System.out.println("The volume of the sphere is: " + V + " cubic meters");`

8) Write a Java program that prompts the user to

enter their first name, middle name, last name, and their preferred title (Mr., Mrs., Ms., Dr., and so on) and displays them formatted in all the following ways.

Title FirstName MiddleName LastName FirstName MiddleName LastName LastName, FirstName LastName, *FirstName MiddleName LastName*, *FirstName MiddleName*, *Title FirstName LastName* For example, assume that the user enters the following: First name: Aphrodite Middle name: Maria Last name: Boura Title: Ms.

The program must display the user's name formatted in all the following ways: Ms. Aphrodite Maria Boura Aphrodite Maria Boura Boura, Aphrodite Boura, Aphrodite Maria Boura, Aphrodite Maria, Ms.

Aphrodite Boura

9) Write a Java program that prompts the user to enter a value for diameter, and then calculates and displays the radius, the perimeter, and the area of a circle. For the same diameter, it must also display the volume of a sphere.

10) Write a Java program that prompts the user to enter the charge for a meal in a restaurant, and then calculates and displays the amount of a 10% tip, 7% sales tax, and the total of all three amounts.

11) A car starts from rest and moves with a constant acceleration along a straight horizontal road for a specified time. Write a Java program that prompts the user to enter the distance traveled as well as the minutes and the seconds traveled, and then calculates the acceleration. The required formula is $S = u_o + \frac{1}{2}at^2$

where

- ***S*** is the distance the car traveled, in meters (m)
- $\boldsymbol{u_o}$ is the initial velocity (speed) of the car, in meters per second (m/sec) ***t*** is the time the car traveled, in seconds (sec) ***a*** is the acceleration, in meters per second^2 (m/sec^2))

12) Write a Java program that prompts the user to enter a temperature in degrees Fahrenheit, and then converts it into its degrees Celsius[10] equivalent. The required formula is

$$\frac{C}{5} = \frac{F - 32}{9}$$

13) The Body Mass Index (BMI) is often used to determine whether a person is overweight or underweight for their height. The formula used to calculate the BMI is $BMI = \frac{weight \cdot 703}{height^2}$

Write a Java program that prompts the user to enter their weight (in pounds) and height (in inches), and then calculates and displays the user's BMI.

14) Write a Java program that prompts the user to enter the subtotal and gratuity rate (on a scale of 0 to 100) and then calculates the tip and total. For example if the user enters 30 and 10, the Java program must display “Tip is $3.00 and total is $33.00”.

15) An employee needs a program to enter the before-tax price of three products and then calculate the final after-tax price of each product, as well as their average

value. Assume a value added tax (VAT) rate of 20%.

16) An employee needs a program to enter the after-tax price of a product, and then calculate its before-tax price. Assume a value added tax (VAT) rate of 20%.

17) Write a Java program that prompts the user to enter the initial price of an item and the discount rate offered (on a scale of 0 to 100), and then calculates and displays the final price and the amount of money saved.

18) Write a Java program that prompts the user to enter the electric meter reading in kilowatt-hours (kWh) at the beginning and end of a month. The program must calculate and display the amount of kWh consumed and the amount of money that must be paid given a cost of each kWh of $0.06 and a value added tax (VAT) rate of 20%.

19) A yacht factory manager needs a program to calculate the profit or loss the factory makes during the period of one year. Here's some information: It costs the factory $1,000,000 to build a yacht.
    - Yachts are sold for $1,500,000 each.
    - The factory pays $250,000 for insurance each month.

Write a Java program that prompts the user to enter the number of yachts sold and then, it calculates and displays the total profit or loss as a positive or negative value correspondingly.

20) Write a Java program that prompts the user to enter two numbers, which correspond to current month and current day of the month, and then calculates and displays the number of days that have elapsed since the beginning of the year. Assume that each month has 30 days.

21) Write a Java program that prompts the user to enter two numbers, which correspond to current month and current day of the month, and then calculates and displays the number of days until the end of the year. Assume that each month has 30 days.

Chapter 11
Manipulating Numbers

11.1 Introduction

Just like every high-level programming language, Java provides many ready-to-use *subprograms* (called methods) that you can use whenever and wherever you wish.

✎ *A "subprogram" is simply a group of statements packaged as a single unit. Each subprogram has a descriptive name and performs a specific task.*

To better understand Java's methods, let's take Heron's[11] iterative formula that calculates the square root of a positive number.

$$x_{n+1} = \frac{\left(x_n + \frac{y}{x_n}\right)}{2}$$

where

- ***y*** is the number for which you want to find the square root ***x_n*** is the *n*-th iteration value of the square root of *y* You might feel a bit frustrated right now. You could think that you should write a program to calculate Heron's formula to find the square root of a number, but this is not true! At present, no one calculates the square root of a number this way. Fortunately, Java includes a method for that purpose! This method, actually a small subprogram, has been given the name `Math.sqrt`, and all you have to do is call it by its name, and it will do the job for you. The `Math.sqrt` method probably uses Heron's iterative formula or perhaps a formula from another ancient or modern mathematician. The truth is that you don't really care! What really matters is that `Math.sqrt` gives you the right result! An example is shown here.

```
x = Double.parseDouble(cin.nextLine()); y = Math.sqrt(x);
System.out.println(y);
```

  Even though Java supports many mathematical subprograms (methods), this chapter covers only those absolutely necessary for this book's purpose. However, if you need even more information

you can visit one of the following addresses:
https://tinyurl.com/y5x2v6w7

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Math.html

✎ *Mathematical subprograms are used whenever you need to perform math calculations, such as finding the square root, sine, cosine, absolute value, and so on.*

11.2 Useful Mathematical Methods (Subprograms), and More

Absolute value

Math.**abs**(*number*)

This method returns the absolute value of `number`.

Example Project_11.2a

```
public static void main(String[] args) throws Exception {
    int a, b;
    a = -5;
    b = Math.abs(a); System.out.println(Math.abs(a)); //It displays: 5
    System.out.println(b); //It displays: 5
    System.out.println(Math.abs(-5.2)); //It displays: 5.2
    System.out.println(Math.abs(5.2)); //It displays: 5.2
}
```

Pi

```
Math.PI
```

This contains the value of π.

✎ *Note that* `Math.PI` *is a constant, not a method. This is why no parentheses are used.*

Example Project_11.2b

```
public static void main(String[] args) throws Exception {
    System.out.println(Math.PI); //It displays: 3.141592653589793
}
```

Sine

```
Math.sin(number)
```

This method returns the sine of *number*. The value of *number* must be expressed in radians. You can multiply by `Math.PI / 180` to convert degrees to radians.

Example Project_11.2c

```
public static void main(String[] args) throws Exception {
    double a, b;
    a = Math.sin(3 * Math.PI / 2); //Sine of 3π/2 radians b = Math.sin(270
    * Math.PI / 180); //Sine of 270 degrees System.out.println(a + " " + b);
    //It displays: -1.0 -1.0
}
```

Cosine

```
Math.cos(number)
```

This method returns the cosine of *number*. The value of *number* must be expressed in radians. You can multiply by `Math.PI / 180` to convert degrees to radians.

Example Project_11.2d

```
public static void main(String[] args) throws Exception {
    double a, b;
    a = Math.cos(2 * Math.PI); //Cosine of 2π radians b = Math.cos(360 *
    Math.PI / 180); //Cosine of 360 degrees System.out.println(a + " " +
    b); //It displays: 1.0 1.0
}
```

Tangent

```
Math.tan(number)
```

This method returns the tangent of *number*. The value of *number* must be expressed in radians. You can multiply by `Math.PI / 180` to convert degrees to radians.

Example Project_11.2e

```
public static void main(String[] args) throws Exception {
    double a;
    a = Math.tan(10 * Math.PI / 180); //Tangent of 10 degrees
    System.out.println(a); //It displays: 0.17632698070846498
}
```

String to integer

Integer.**parseInt**(*value*)

This method converts a string representation of an integer to its numeric equivalent.

Example Project_11.2f

```
public static void main(String[] args) throws Exception {
    String s1 = "5"; String s2 = "3"; int k;
    k = Integer.parseInt(s1); System.out.println(k); //It displays: 5
    System.out.println(Integer.parseInt(s2)); //It displays: 3
    System.out.println(s1 + s2); //It displays: 53
    System.out.println(Integer.parseInt(s1) + Integer.parseInt(s2)); //It
    displays: 8
}
```

String to real

Double.**parseDouble**(*value*)

This method converts a string representation of a real to its numeric equivalent.

Example Project_11.2g

```
public static void main(String[] args) throws Exception {
    String s1 = "6.5"; String s2 = "3.4"; double x;
    x = Double.parseDouble(s1); System.out.println(x); //It displays: 6.5
    System.out.println(Double.parseDouble(s2)); //It displays: 3.4
    System.out.println(s1 + s2); //It displays: 6.53.4
    System.out.println(Double.parseDouble(s1) +
    Double.parseDouble(s2)); //It displays: 9.9
}
```

Integer value (Type casting)

(**int**)*number*

This returns the integer portion of *number*. If *number* contains a fractional part, that part is lost during conversion.

Example Project_11.2h

```
public static void main(String[] args) throws Exception {
    double a = 5.4;
    System.out.println((int)a); //It displays: 5
    System.out.println((int)34); //It displays: 34
    System.out.println((int)34.9); //It displays: 34
    System.out.println((int)-34.999); //It displays: -34
}
```

> ✎ *In computer science, type casting is a way of converting a variable of one data type into another. Note that* `(int)` *is not a method. It is just a way in Java to turn a real into an integer. Also note that if a real contains a fractional part, that part is lost during conversion.*

Real value (Type casting)

(**double**)*number*

This returns the *number* as real.

Example Project_11.2i

```
public static void main(String[] args) throws Exception {
    int a = 5;
    System.out.println((double)a); //It displays: 5.0
    System.out.println((double)34); //It displays: 34.0
    System.out.println((double)-34); //It displays: -34.0
    System.out.println(a / 2); //It displays: 2
    System.out.println((double)a / 2); //It displays: 2.5
    System.out.println((double)a / 2); //It displays: 2.5
    System.out.println(a / (double)2); //It displays: 2.5
    System.out.println((double)a / (double)2); //It displays: 2.5
    System.out.println(a / 2.0); //It displays: 2.5
}
```

> ✎ *In computer science, type casting is a way of converting a variable of one data type into another. Note that* `(double)` *is not a method. It is just a way in Java to turn an integer into a real (float).*

✎ *In Java, the result of the division of two integers is always an integer. Thus, in the expression* `a / 2` *of the previous example, since both variable* `a` *and number 2 are integers, the results is an integer. If you wish a result of type* `double`*, at least one of the operands of the division must be of type* `double`**.**

Power

```
Math.pow(number, exp)
```

This method returns the result of *number* raised to the power of *exp*.

Example Project_11.2j

```
public static void main(String[] args) throws Exception {
    double a, b, c;
    a = 2;
    b = 3;
    System.out.println(Math.pow(a, b)); //It displays: 8.0
    c = Math.pow(3, 2); System.out.println(c); //It displays: 9.0
}
```

✎ *The method* `Math.pow()` *serves a dual role. Apart from being used to calculate the power of a value raised to another value, it is also used to compute any root of a number using the known mathematical formula* $\sqrt[z]{X} = X^{\frac{1}{z}}$*. For example, you can write* `y = Math.pow(x, 1 / 2.0)` *to calculate the square root of* `x` *or* `y = Math.pow(x, 1 / 3.0)` *to calculate the cubic root of* `x`*!*

Random

```
Math.random()
```

This method returns a pseudo-random number (real) in the range [0, 1).

✎ *Note that this range does not include the 1.*

If you want a pseudo-random **integer** between *minimum* and *maximum* you can use the following formula `minimum + (int)(Math.random() * (maximum - minimum + 1))` Example Project_11.2k

```
public static void main(String[] args) throws Exception {
    //Display a random double in the range [0, 1)
    System.out.println(Math.random());
```

```
    //Display a random integer between 2 and 10
    System.out.println(2 + (int)(Math.random() * (10 - 2 + 1)));
    //Display a random integer between 20 and 25
    System.out.println(20 + (int)(Math.random() * 6)); }
```

✎ *Random numbers are widely used in computer games. For example, an "enemy" may show up at a random time or move in random directions. Also, random numbers are used in simulation programs, in statistical programs, in computer security to encrypt data, and so on.*

Round

`Math.round(number)`

This method returns the closest integer of `number`.

Example Project_11.2l

```
public static void main(String[] args) throws Exception {
    double a = 5.9;
    System.out.println(Math.round(a)); //It displays: 6
    System.out.println(Math.round(5.4)); //It displays: 5
}
```

If you need the rounded value of `number` to a specified `precision`, you can use the following formula: `Math.round(number * Math.pow(10, precision)) / Math.pow(10, precision)` Example Project_11.2m

```
public static void main(String[] args) throws Exception {
    double a, y;
    a = 5.312;
    y = Math.round(a * Math.pow(10, 2)) / Math.pow(10, 2);
    System.out.println(y); //It displays: 5.31
    a = 5.315;
    y = Math.round(a * Math.pow(10, 2)) / Math.pow(10, 2);
    System.out.println(y); //It displays: 5.32
    //Display 2.345
    System.out.println(Math.round(2.3447 * Math.pow(10, 3)) /
    Math.pow(10, 3));
    //Display 2.345
    System.out.println(Math.round(2.3447 * 1000) / 1000); }
```

Square root

`Math.sqrt(number)`

This method returns the square root of *number*, where *number* can be a positive value or zero.

Example Project_11.2n

```
public static void main(String[] args) throws Exception {
    double x, y;
    System.out.println(Math.sqrt(9)); //It displays: 3.0
    System.out.println(Math.sqrt(2)); //It displays: 1.4142135623730951
    x = Math.sqrt(8); System.out.println(x); //It displays:
    2.8284271247461903
    y = Math.round(Math.sqrt(8)); System.out.println(y); //It displays: 3
}
```

✎ *Note how the method* `Math.sqrt()` *is nested within the method* `Math.round()`. *The result of the inner (nested) method (or methods) is used as an argument for the outer method. This is a writing style that most programmers prefer to follow because it helps to save a lot of code lines. Of course, if you nest too many methods, no one will be able to understand your code. A nesting of up to four levels is quite acceptable.*

Exercise 11.2-1 Calculating the Distance Between Two Points

Write a Java program that prompts the user to enter the coordinates (x, y) of two points and then calculates the straight line distance between them. The required formula is $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Solution In this exercise, you need to use the method `Math.sqrt()`, which returns the square root of a number.

To simplify things, the terms $(x_1 - x_2)^2$ and $(y_1 - y_2)^2$ are calculated individually and the results are assigned to two temporary variables. The Java program is shown here.

Project_11.2-1a

```
public static void main(String[] args) throws Exception
                                {
        double d, x1, x2, xTemp, y1, y2, yTemp;
     System.out.println("Enter coordinates for point
   A:"); x1 = Double.parseDouble(cin.nextLine()); y1 =
```

```
        Double.parseDouble(cin.nextLine());
    System.out.println("Enter coordinates for point
  B:"); x2 = Double.parseDouble(cin.nextLine()); y2 =
        Double.parseDouble(cin.nextLine());
  xTemp = Math.pow(x1 - x2, 2); yTemp = Math.pow(y1 -
                        y2, 2);
          d = Math.sqrt(xTemp + yTemp);
 System.out.println("Distance between points: " + d);
                          }
```

Now, let's see another approach. It is actually possible to perform all operations within the method call. Doing that, the result of the operations will be used as an argument for the method. This is a writing style that most programmers prefer to follow because it can save a lot of variables and code lines. The Java program is shown here.

```
                  Project_11.2-1b
public static void main(String[] args) throws Exception
                          {
              double d, x1, x2, y1, y2;
    System.out.println("Enter coordinates for point
  A:"); x1 = Double.parseDouble(cin.nextLine()); y1 =
        Double.parseDouble(cin.nextLine());
    System.out.println("Enter coordinates for point
  B:"); x2 = Double.parseDouble(cin.nextLine()); y2 =
        Double.parseDouble(cin.nextLine());
   d = Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 -
     y2, 2)); System.out.println("Distance between
                  points: " + d); }
```

You can nest one subprogram within another. Note how the methods `Math.pow()` *are nested within the method* `Math.sqrt()`*. The result of the inner (nested) methods is used as an argument for the outer method.*

Exercise 11.2-2 How Far Did the Car Travel?

A car starts from rest and moves with a constant acceleration along a straight horizontal road for a specified distance. Write a Java program that prompts the user to enter the acceleration and the distance the car

traveled and then calculates the time traveled. The required formula is

$$S = u_o + \frac{1}{2}at^2$$

*where ► **S** is the distance the car traveled, in meters (m) ► **u_o** is the initial velocity (speed) of the car, in meters per second (m/sec) ► **t** is the time the car traveled, in seconds (sec) ► **a** is the acceleration, in meters per second^2 (m/sec^2) Solution Since the car starts from rest, the initial velocity (speed) u_0 is zero. Thus, the formula becomes* $S = \frac{1}{2}at^2$

Now, if you solve for time, the final formula becomes $t = \sqrt{\frac{2S}{a}}$

In Java, you can use the `Math.sqrt()` method, which returns the square root of a number.

Project_11.2-2

```
public static void main(String[] args) throws Exception {
   double S, a, t;
   System.out.print("Enter acceleration: "); a = Double.parseDouble(cin.nextLine());
   System.out.print("Enter distance traveled: "); S =
   Double.parseDouble(cin.nextLine());
   t = Math.sqrt(2 * S / a);
   System.out.println("Your car traveled for " + t + " seconds"); }
```

11.3 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Java methods are small subprograms that solve small problems.
2) Every programmer must use Heron's iterative formula to calculate the square root of a positive number.
3) The `Math.abs()` method returns the absolute position of an item.
4) The statement `(int)3.59` returns a result of 3.6.
5) The statement `y = (int)"two"` is a valid Java statement.
6) The statement `y = Integer.parseInt("2")` is a valid Java statement.
7) The statement `(int)3` returns a result of 3.0.
8) The statement `(double)3` returns a result of 3.0.

9) The statement `y = Double.parseDouble("3.14")` is not a valid Java statement.
10) The `Math.PI` constant is equal to 3.14.
11) The statement `Math.pow(2, 3)` returns a result of 6.
12) The statement `Math.pow(2, 3)` returns a result of 9.
13) The `Math.random()` method returns a random real (float).
14) There is a 50% possibility that the statement `y = (int) (Math.random() * 2)` will assign a value of 1 to variable `y`.
15) The statement `Math.round(3.59)` returns a result of 4.
16) To calculate the sine of 90 degrees, you have to write `y = Math.sin(Math.PI / 2)`
17) The statement `y = Math.sqrt(-2)` is valid.
18) The following code fragment satisfies the property of definiteness.

```
double a, b, x; a = Double.parseDouble(cin.nextLine()); b =
Double.parseDouble(cin.nextLine()); x = a * Math.sqrt(b);
System.out.println(x);
```

11.4 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) Which of the following calculates the result of the variable a raised to the power of 2?
   a) `y = a * a;`
   b) `y = Math.pow(a, 2);`
   c) `y = a * a / a * a;`
   d) all of the above
2) What is the value of the variable `y` when the statement `y = Math.abs(+5.2)` is executed?
   a) −5.2
   b) −5
   c) 0.2
   d) 5.2
   e) none of the above
3) Which of the following calculates the sine of 180 degrees?
   a) `Math.sin(180)`
   b) `Math.sin(Math.PI)`
   c) all of the above
   d) none of the above
4) What is the value of the variable `y` when the statement `y = (int)(5.0 / 2.0)` is executed?
   a) 2.5

b) 3

c) 2

d) 0.5

5) What is the value of the variable `y` when the statement `y = Math.pow(Math.sqrt(4), 2)` is executed?

a) 4

b) 2

c) 8

d) 16

6) What is the value of the variable `y` when the statement `y = Math.round(5.2) / 2.0` is executed?

a) 2

b) 2.5

c) 2.6

d) none of the above

11.5 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step of the Java program for two different executions.

   The input values for the two executions are: (i) 9, and (ii) 4.

```
public static void main(String[] args) throws Exception {
   double a, b, c;
   a = Double.parseDouble(cin.nextLine()); a += 6 / Math.sqrt(a) * 2 + 20.4; b
   = Math.round(a) % 4; c = b % 3;
   System.out.println(a + ", " + b + ", " + c); }
```

2) Create a trace table to determine the values of the variables in each step of the Java program for two different executions.

   The input values for the two executions are: (i) −2, and (ii) −3

```
public static void main(String[] args) throws Exception {
   int a, b, c;
   a = Integer.parseInt(cin.nextLine());
   b = Math.abs(a) % 4 + (int)Math.pow(a, 4); c = b % 5;
   System.out.println(b + ", " + c); }
```

3) Write a Java program that prompts the user to enter an angle θ in radians and then calculates and displays the angle in degrees. It is given that $2\pi = 360^{o}$.

4) Write a Java program that prompts the user to enter the two right angle sides A and B of a right-angled triangle and then calculates its hypotenuse. It is known from the Pythagorean[12] theorem that $hypotenuse = \sqrt{A^2 + B^2}$

5) Write a Java program that prompts the user to enter the angle θ (in degrees) of a right-angled triangle and the length of its adjacent side, and then calculates the length of the opposite side. It is known that $\tan(\theta) = \frac{Opposite}{Adjacent}$

Chapter 12
Complex Mathematical Expressions

12.1 Writing Complex Mathematical Expressions

In Section 7.2 you learned all about arithmetic operators but little about how to use them and how to write your own complex mathematical expressions. In this chapter, you are going to learn how easy is to convert mathematical expressions to Java statements.

Arithmetic operators follow the same precedence rules as in mathematics, which means that multiplication and division are performed first, and addition and subtraction are performed next. Moreover, when multiplication and division co-exist in the same expression, and since both are of the same precedence, these operations are performed left to right.

The method `Math.pow()` *serves a dual role. Apart from being used to calculate the power of a value raised to another value, it is also used to compute any root of a number using the known mathematical formula* $\sqrt[z]{X} = X^{\frac{1}{z}}$. *For example, you can write* `y = Math.pow(x, 1 / 3.0)` *to calculate the cubic root of* `x` *or* `y = Math.pow(x, 1 / 5.0)` *to calculate the fifth root of* `x`*!*

Exercise 12.1-1 Representing Mathematical Expressions in Java

Which of the following Java statements correctly represent the following mathematical expression?

$$x = \frac{1}{10 + z} 27$$

i) `x = 1 * 27 / 10 + z;`

ii) `x = 1 · 27 / (10 + z);`

iii) `x = 27 / 10 + z;`

iv) `x = 27 / (10 + z);`

v) `x = (1 / 10 + z) * 27;`

vi) `x = 1 / ((10 + z) * 27);`

vii) `x = 1 / (10 + z) * 27;`

viii) `x = 1 / (10 + z) / 27;`

Solution i) Wrong. Since the multiplication and the division are performed before the addition, this is equivalent to $x = \frac{1 \cdot 27}{10} + z$.

ii) **Wrong**. An asterisk must have been used for multiplication.

iii) **Wrong**. Since the division is performed before the addition, this is equivalent to $x = \frac{27}{10} + z$.

iv) **Correct**. This is equivalent to $x = \frac{27}{10 + z}$.

v) **Wrong**. Inside parentheses, the division is performed before the addition. This is equivalent to $x = \left(\frac{1}{10} + z\right) 27$.

vi) **Wrong**. Parentheses are executed first and this is equivalent to $x = \frac{1}{(10+z)27}$.

vii) **Correct**. Division is performed before multiplication (left to right). The term $\frac{1}{10+z}$ is calculated first and then, the result is multiplied by 27.

viii) **Wrong**. This is equivalent to $x = \frac{\frac{1}{10+z}}{27}$.

Exercise 12.1-2 Writing a Mathematical Expression in Java

Write a Java program that calculates the mathematical expression $y = 10\,x - \frac{10 - z}{4}$

Solution First, you must distinguish between the data input and the output result. Obviously, the output result is assigned to y and the user must enter values for x and z. The solution for this exercise is shown here.

Project_12.1-2

```
public static void main(String[] args) throws Exception {
   double x, y, z;
   x = Double.parseDouble(cin.nextLine()); z = Double.parseDouble(cin.nextLine());
   y = 10 * x - (10 - z) / 4; System.out.println("The result is: " + y); }
```

Exercise 12.1-3 Writing a Complex Mathematical Expression in Java

Write a Java program that calculates the mathematical expression

$$y = \frac{5\dfrac{3x^2 + 5x + 2}{7w + \dfrac{1}{z}} + z}{4\dfrac{3 + x}{7}}$$

Assume that the user enters only positive values for x, w, and z.

Solution Oops! Now the expression is more complex! In fact, it is much more complex! So, let's take a look at a quite different approach. The main idea is to break the complex expression into smaller, simpler expressions and assign each sub-result to temporary variables. In the end, you can build the original expression out of all these temporary variables! This approach is presented next.

```
                    Project_12.1-3a
public static void main(String[] args) throws Exception {
   double denominator, nominator, temp1, temp2, temp3, w,
                              x, y, z;
         x = Double.parseDouble(cin.nextLine()); w =
           Double.parseDouble(cin.nextLine()); z =
             Double.parseDouble(cin.nextLine());
    temp1 = 3 * x * x + 5 * x + 2; temp2 = 7 * w + 1 / z;
     temp3 = (3 + x) / 7; nominator = 5 * temp1 / temp2 +
                  z; denominator = 4 * temp3;
     y = nominator / denominator; System.out.println("The
                    result is: " + y); }
```

You may say, "*Okay, but I wasted so many variables and as everybody knows, each variable is a portion of main memory. How can I write the original expression in one single line and waste less memory?*"

This job may be a piece of cake for an advanced programmer, but what about you? What about a novice programmer?

The next method will help you write even the most complex mathematical expressions without any syntax or logic errors! The rule is very simple. "*After breaking the complex expression into smaller, simpler expressions and assigning each sub-result to temporary variables, start backwards and replace each variable with its assigned expression. Be careful though! When*

you replace a variable with its assigned expression, you must always enclose the expression in parentheses!"

Confused? Don't be! It's easier in action. Let's try to rewrite the previous Java program. Starting backwards, replace variables `nominator` and `denominator` with their assigned expressions. The result is

```
y = (5 * temp1 / temp2 + z) / (4 * temp3)
```

> ✎ *Note the extra parentheses added.*

Now you must replace variables `temp1`, `temp2`, and `temp3` with their assigned expressions, and the one-line expression is complete!

```
y = (5 * (3 * x ** 2 + 5 * x + 2) / (7 * w + 1 / z) + z) / (4 * ((3 + x) / 7))
```

It may look scary at the end but it wasn't that difficult, was it?

The Java program can now be rewritten Project_12.1-3b

```
public static void main(String[] args) throws Exception {
    double w, x, y, z;
    x = Double.parseDouble(cin.nextLine()); w =
    Double.parseDouble(cin.nextLine()); z =
    Double.parseDouble(cin.nextLine());
    y = (5 * (3 * x * x + 5 * x + 2) / (7 * w + 1 / z) + z) / (4 * ((3 + x) / 7));
    System.out.println("The result is: " + y); }
```

12.2 Review Exercises

Complete the following exercises.

1) Match each element from the first table with one **or more** elements from the second table.

Expression
i) `5 / Math.pow(x, 2) * y + Math.pow(x, 3)`
ii) `5 / (Math.pow(x, 3) * y) + Math.pow(x, 2)`

Expression
a) `5 * y / Math.pow(x, 2) + Math.pow(x, 3)`
b) `5 * y / x * x + Math.pow(x, 3)`
c) `5 / (x * x * x * y) + x * x`
d) `5 / (x * x * x) * y + x * x`
e) `5 * y / (x * x) + x * x * x`
f) `1 / (x * x * x * y) * 5 + x * x`
g) `y / (x * x) * 5 + Math.pow(x, 3)`
h) `1 / (x * x) * 5 * y + x / 1 * x * x`

2) Write the following mathematical expressions in Java using one line of code for each.

i) $y = \frac{(x+3)^{5w}}{7(x-4)}$

ii) $y = \sqrt[5]{\left(3x^2 - \frac{1}{4}x^3\right)}$

iii) $y = \frac{\sqrt{x^4-2x^3-7x^2+x}}{\sqrt[3]{4\left(7x^4-\frac{3}{4}x^3\right)(7x^2+x)}}$

iv) $y = \frac{x}{x-3(x-1)} + \left(x\sqrt[5]{x-1}\right)\frac{1}{(x^3-2)(x-1)^3}$

v) $y = \left(\sin\left(\frac{\pi}{3}\right) - \cos\left(\frac{\pi}{2}w\right)\right)^2$

vi) $y = \frac{\left(\sin\left(\frac{\pi}{2}x\right)+\cos\left(\frac{3\pi}{2}w\right)\right)^3}{\left(\tan\left(\frac{2\pi}{3}w\right)-\sin\left(\frac{\pi}{2}x\right)\right)^{\frac{1}{2}}} + 6$

3) Write a Java program that prompts the user to enter a value for *x* and then calculates and displays the result of the following mathematical

expression.

$$y = \sqrt{x^2 + 1}(x^3 + x^2)$$

4) Write a Java program that prompts the user to enter a value for *x* and then calculates and displays the result of the following mathematical expression.

$$y = \frac{7x}{2x + 4(x^2 + 4)}$$

Suggestion: Try to write the expression in one line of code.

5) Write a Java program that prompts the user to enter a positive value for *x* and *w* and then calculates and displays the result of the following mathematical expression.

$$y = \frac{x^{x+1}}{\left(\tan\left(\frac{2w}{3} + 5\right) + \tan\left(\frac{x}{2} + 1\right)\right)^3}$$

Suggestion: Try to write the expression in one line of code 6) Write a Java program that prompts the user to enter a positive value for *x* and *w* and then calculates and displays the result of the following mathematical expression.

$$y = \frac{3 + w}{6x + 7(x + 4)} + \left(x\sqrt[5]{3w + 1}\right)\frac{5x + 4}{(x^3 + 3)(x - 1)^6}$$

Suggestion: Try to write the expression in one line of code.

7) Write a Java program that prompts the user to enter a positive value for *x* and *w* and then calculates and displays the result of the following mathematical expression.

$$y = \frac{x^x}{\left(\sin\left(\frac{2w}{3} + 5\right) - x\right)^2} + \frac{(\sin(3x) + \mathrm{w})^{x+1}}{\left(\sqrt{7w}\right)^{\frac{3}{2}}}$$

Suggestion: Try to write the expression in one line of code 8) Write a Java program that prompts the user to enter the lengths of all three sides A, B, and C, of a triangle and then calculates and displays the area of the triangle. You can use Heron's formula, which has been known for nearly 2,000 years!

$$Area = \sqrt{S(S - A)(S - B)(S - C)}$$

where *S* is the semi-perimeter $S = \frac{A+B+C}{2}$

Chapter 13
Exercises With a Quotient and a Remainder

13.1 Introduction

What types of problems might require the use of the quotient and the remainder of an integer division? While there may not be a simple answer to this question, quotients and remainders can be used to: ► split a number into individual digits ► examine if an integer is odd or even ► examine if a number is a multiple of another number ► convert an elapsed time (in seconds) to hours, minutes, and seconds ► convert an amount of money (in USD) to a number of $100 notes, $50 notes, $20 notes, and such ► calculate the greatest common divisor ► determine if a number is a palindrome ► count the number of digits within a number ► determine how many times a specific digit occurs within a number Of course, these are some of the uses and certainly you can find so many others. Next you will see some exercises that make use of the quotient and the remainder of integer division.

Exercise 13.1-1 Calculating the Quotient and Remainder of Integer Division

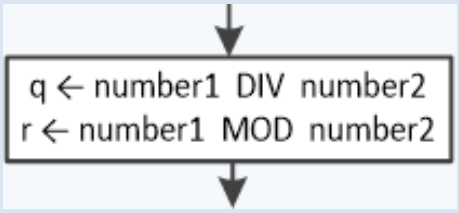
Write a Java program that prompts the user to enter two integers and then calculates the quotient and the remainder of the integer division.

Solution The modulus (%) operator performs an integer division and returns the integer remainder. Since Java doesn't actually incorporate an arithmetic operator that calculates the integer quotient, you can use the `(int)` casting operator to achieve the same result. The solution is presented here.

Project_13.1-1

```
public static void main(String[] args) throws Exception {
   int number1, number2, q, r;
   System.out.print("Enter first number: "); number1 = Integer.parseInt(cin.nextLine());
   System.out.print("Enter second number: "); number2 = Integer.parseInt(cin.nextLine());
   q = (int)(number1 / number2); r = number1 % number2; System.out.println("Integer Quotient: " +
   q + "\nInteger Remainder: " + r); }
```

✎ *In flowcharts, in order to calculate the quotient and the remainder of an integer division, you can use the popular* `DIV` *and* `MOD` *operators. An example is shown here.*



✎ *In Java, the result of the division of two integers is always an integer. Thus, in the statement* `q = (int)(number1 / number2)`*, since variables* `number1` *and* `number2` *are integers, the*

`(int)` *casting operator is redundant. However, it is a good practice to keep it there just for improved readability.*

Exercise 13.1-2 Finding the Sum of Digits

Write a Java program that prompts the user to enter a four-digit integer and then calculates the sum of its digits.

Solution What you should keep in mind here is that statements like this one

number = Integer.**parseInt**(cin.**nextLine**());

assign the user-provided four-digit integer to one single variable, `number`, and not to four individual variables. So, after the user enters the four-digit integer, the program must split the integer into its four digits and assign each digit to a separate variable. Then it can calculate the sum of these four variables and get the required result. There are two approaches available.

First approach Let's try to understand the first approach using an arithmetic example. Take the number 6753, for example.

First digit = 6	The **first digit** can be isolated if you divide the user-provided number by 1000 using the (/) operator and the `(int)` type casting operator to get the integer quotient `digit1 = (int)(6753 / 1000)`
Remaining digits = 753	The remaining digits can be isolated if you divide the user-provided number by 1000 again, this time using the (%) operator to get the integer remainder `r = 6753 % 1000`
Second digit = 7	The **second digit** can be isolated if you divide the remaining digits by 100 using the (/) operator and the `(int)` type casting operator to get the integer quotient `digit2 = (int)(753 / 100)`
Remaining digits = 53	The remaining digits are now `r = 753 % 100`
Third digit = 5	The **third digit** can be isolated if you divide the remaining digits by 10 using the (/) operator and the `(int)` type casting operator to get the integer quotient `digit3 = (int)(53 / 10)`
Fourth digit = 3	The last remaining digit, which happens to be the **fourth digit**, is `digit4 = 53 % 10`

The Java program that solves this algorithm is shown here.

Project_13.1-2a

```
public static void main(String[] args) throws Exception {
   int digit1, digit2, digit3, digit4, number, r, total;
 System.out.print("Enter a four-digit integer: "); number =
              Integer.parseInt(cin.nextLine());
      digit1 = (int)(number / 1000); r = number % 1000;
           digit2 = (int)(r / 100); r = r % 100;
          digit3 = (int)(r / 10); digit4 = r % 10;
         total = digit1 + digit2 + digit3 + digit4;
              System.out.println(total); }
```

The trace table for the program that you have just seen is shown here.

Step	Statement	Notes	number	digit1	digit2	digit3	digit4	r	total
1	`.print("Enter…`	It displays: Enter a four-digit integer:							
2	`number = Integer.parseInt(…`	User enters 6753	**6753**	?	?	?	?	?	?
3	`digit1 = (int) (number / 1000)`		6753	**6**	?	?	?	?	?
4	`r = number % 1000`		6753	6	?	?	?	**753**	?
5	`digit2 = (int)(r / 100)`		6753	6	**7**	?	?	753	?
6	`r = r % 100`		6753	6	7	?	?	**53**	?
7	`digit3 = (int)(r / 10)`		6753	6	7	**5**	?	53	?
8	`digit4 = r % 10`		6753	6	7	5	**3**	53	?
9	`total = digit1 + digit2 + digit3 + digit4`		6753	6	7	5	3	53	**21**
10	`.println(total)`	It displays: 21							

To further help you, find below a general purpose Java program that can be used to split any given integer. Since the length of your program depends on the number of digits, `N`, all you have to do is write `N−1` pairs of statements.

```
System.out.print("Enter an N-digit integer: "); number = Integer.parseInt(cin.nextLine());
digit1 = (int)(number / 10^(N-1)); r = number % 10^(N-1);
digit2 = (int)(r / 10^(N-2)); r = r % 10^(N-2); .
.
.
```

```
digit(N-2) = (int)(r / 100); r = r % 100;
digit(N-1) = (int)(r / 10); digit(N) = r % 10;
```

For example, if you want to split a six-digit integer, you need to write five pairs of statements as shown in the program that follows.

```
Project_13.1-2b
public static void main(String[] args) throws Exception {
 int digit1, digit2, digit3, digit4, digit5, digit6, number,
                                                        r;
  System.out.print("Enter an six-digit integer: "); number =
                    Integer.parseInt(cin.nextLine());
     digit1 = (int)(number / 100000); r = number % 100000;
           digit2 = (int)(r / 10000); r = r % 10000;
            digit3 = (int)(r / 1000); r = r % 1000;
             digit4 = (int)(r / 100); r = r % 100;
           digit5 = (int)(r / 10); digit6 = r % 10;
System.out.print(digit1 + " " + digit2 + " " + digit3 + " ");
 System.out.println(digit4 + " " + digit5 + " " + digit6); }
```

Second approach For a four-digit given integer, the first approach performs three pairs of divisions—first by 1000, then by 100, and finally by 10—isolating the digits from left to right. In contrast, the three pairs of divisions in this second approach are all by 10, isolating the digits from right to left. Once again, to delve deeper into this approach, let's use an arithmetic example. Consider the same user-provided number: 6753.

Fourth digit = 3	The **fourth digit** can be isolated if you divide the user-provided number by 10 using the (%) operator to get the integer remainder `digit4 = 6753 % 10`
Remaining digits = 675	The remaining digits can be isolated if you divide the user-provided number by 10 again using the (/) operator and the `(int)` type casting operator to get the integer quotient `r = (int)(6753 / 10)`
Third digit = 5	The **third digit** can be isolated if you divide the remaining digits by 10 using the (%) operator to get the integer remainder `digit3 = 675 % 10`
Remaining digits = 67	The remaining digits are now `r = (int)(675 / 10)`
Second digit = 7	The **second digit** can be isolated if you divide the remaining digits by 10 using the (%) operator to get the integer remainder `digit2 = 67 % 10`
First digit	The last remaining digit, which happens to be the **first digit**, is

= 6	digit1 = (int)(67 / 10)

The Java program for this algorithm is shown here.

Project_13.1-2c

```
public static void main(String[] args) throws Exception {
   int digit1, digit2, digit3, digit4, number, r, total;
 System.out.print("Enter a four-digit integer: "); number =
             Integer.parseInt(cin.nextLine());
       digit4 = number % 10; r = (int)(number / 10);
           digit3 = r % 10; r = (int)(r / 10);
         digit2 = r % 10; digit1 = (int)(r / 10);
        total = digit1 + digit2 + digit3 + digit4;
              System.out.println(total); }
```

To further help you, find below a general purpose Java program that can be used to split any given integer. This program uses the second approach. Once again, since the length of your program depends on the number of the digits, `N`, all you have to do is write `N−1` pairs of statements.

```
System.out.print("Enter a N-digit integer: "); number = Integer.parseInt(cin.nextLine());
digit(N) = number % 10; r = (int)(number / 10);
digit(N-1) = r % 10; r = (int)(r / 10); .
.
.
digit3 = r % 10;
r = (int)(r / 10);
digit2 = r % 10;
digit1 = (int)(r / 10);
```

For example, if you want to split a five-digit integer, you must use four pairs of statements as shown in the program that follows.

Project_13.1-2d

```
public static void main(String[] args) throws Exception {
   int digit1, digit2, digit3, digit4, digit5, number, r;
 System.out.print("Enter a five-digit integer: "); number =
             Integer.parseInt(cin.nextLine());
       digit5 = number % 10; r = (int)(number / 10);
           digit4 = r % 10; r = (int)(r / 10);
           digit3 = r % 10; r = (int)(r / 10);
         digit2 = r % 10; digit1 = (int)(r / 10);
System.out.println(digit1 + " " + digit2 + " " + digit3 + " "
              + digit4 + " " + digit5); }
```

Exercise 13.1-3 Displaying an Elapsed Time

Write a Java program that prompts the user to enter an integer that represents an elapsed time in seconds and then displays it in the format "DD day(s) HH hour(s) MM minute(s) and SS second(s)". For example, if the user enters the number 700005, the message "8 day(s) 2 hour(s) 26 minute(s) and 45 second(s)" must be displayed.

Solution As you may already know, there are 60 seconds in a minute, 3600 seconds in an hour (60 × 60), and 86400 seconds in a day (3600 × 24). Let's try to analyze the number 700005 using the first approach that you learned in the previous exercise.

Days = 8	The number of **days** can be isolated if you divide the user-provided integer by 86400 using the (/) operator and the `(int)` type casting operator to get the integer quotient `days = (int)(700005 / 86400)`
Remaining seconds = 8805	The remaining seconds can be isolated if you divide the user-provided integer by 86400 again, this time using the (%) operator to get the integer remainder `r = 700005 % 86400`
Hours = 2	The number of **hours** can be isolated if you divide the remaining seconds by 3600 using the (/) operator and the `(int)` type casting operator to get the integer quotient `hours = (int)(8805 / 3600)`
Remaining seconds = 1605	The remaining seconds are now `r = 8805 % 3600`
Minutes = 26	The number of **minutes** can be isolated if you divide the remaining seconds by 60 using the (/) operator and the `(int)` type casting operator to get the integer quotient `minutes = (int)(1605 / 60)`
Seconds = 45	The last remainder, which happens to be the number of **seconds** left, is `seconds = 1605 % 60`

The Java program for this algorithm is as follows.

Project_13.1-3a

```
public static void main(String[] args) throws Exception {
        int days, hours, minutes, number, r, seconds;
    System.out.print("Enter a period of time in seconds: ");
        number = Integer.parseInt(cin.nextLine());
     days = (int)(number / 86400); // 60 * 60 * 24 = 86400
                    r = number % 86400;
        hours = (int)(r / 3600); // 60 * 60 = 3600
                        r = r % 3600;
```

```
        minutes = (int)(r / 60); seconds = r % 60;
    System.out.println(days + " day(s) " + hours + " hour(s) ");
    System.out.println(minutes + " minute(s) and " + seconds + "
                        second(s)"); }
```

You can also solve this exercise using the second approach from the previous exercise. All you have to do is first divide by 60, then divide by 60 again, and finally divide by 24, as shown here.

```
                Project_13.1-3b
    public static void main(String[] args) throws Exception {
            int days, hours, minutes, number, r, seconds;
      System.out.print("Enter a period of time in seconds: ");
            number = Integer.parseInt(cin.nextLine());
          seconds = number % 60; r = (int)(number / 60);
              minutes = r % 60; r = (int)(r / 60);
             hours = r % 24; days = (int)(r / 24);
    System.out.println(days + " day(s) " + hours + " hour(s) ");
    System.out.println(minutes + " minute(s) and " + seconds + "
                        second(s)"); }
```

Exercise 13.1-4 Reversing a Number

Write a Java program that prompts the user to enter a three-digit integer and then builds and displays its reverse. For example, if the user enters the number 875, the program must display 578.

Solution To isolate the three digits of the user-provided integer, you can use either first or second approach. Afterward, the only difficulty in this exercise is to build the reversed number.

Take the number 875, for example. The three digits, after isolation, will be:

digit1 = 8
digit2 = 7
digit3 = 5

You can then build the reversed number by simply calculating the sum of the products: digit3 × 100 + digit2 × 10 + digit1 × 1 = 5 × 100 + 7 × 10 + 8 × 1 = 578

For a change, let's split the user-provided number using the second approach. The Java program will look like this.

```
                Project_13.1-4
public static void main(String[] args) throws Exception {
   int digit1, digit2, digit3, number, r, reversedNumber;
   System.out.print("Enter a three-digit integer: "); number = Integer.parseInt(cin.nextLine());
   digit3 = number % 10; //This is the rightmost digit r = (int)(number / 10);
```

```
digit2 = r % 10; //This is the digit in the middle digit1 = (int)(r / 10); //This is the
leftmost digit
reversedNumber = digit3 * 100 + digit2 * 10 + digit1; System.out.println(reversedNumber); }
```

13.2 Review Exercises

Complete the following exercises.

1) Write a Java program that prompts the user to enter any integer and then multiplies its last digit by 8 and displays the result.

   Hint: It is not necessary to know the exact number of digits. You can isolate the last digit of any integer using a modulus 10 operation.

2) Write a Java program that prompts the user to enter a five-digit integer. The program must then find and display the sum of the original number and its reverse. For example, if the user enters the number 32675, the program must display the message “32675 + 57623 = 90298”.

3) Write a Java program that prompts the user to enter an integer and then it displays 1 when the number is odd; otherwise, it displays 0. Try not to use any decision control structures since you haven't learned anything about them yet!

4) Write a Java program that prompts the user to enter an integer and then it displays 1 when the number is even; otherwise, it displays 0. Try not to use any decision control structures since you haven't learned anything about them yet!

5) Write a Java program that prompts the user to enter an integer representing an elapsed time in seconds and then displays it in the format “WW week(s) DD day(s) HH hour(s) MM minute(s) and SS second(s)”. For example, if the user enters the number 2000000, the message “3 week(s) 2 day(s) 3 hour(s) 33 minute(s) and 20 second(s)” must be displayed.

6) Inside an ATM bank machine there are notes of $20, $10, $5, and $1. Write a Java program that prompts the user to enter the amount of money they want to withdraw (using an integer value) and then displays the least number of notes the ATM must give. For example, if the user enters an amount of $76, the program must display the message “3 note(s) of $20, 1 note(s) of $10, 1 note(s) of $5, and 1 note(s) of $1”.

7) A robot arrives on the moon in order to perform some experiments. Each of the robot's steps is 25 inches long. Write a Java program that prompts the user to enter the number of steps the robot made and then calculates and displays the distance traveled in miles, feet, yards, and inches. For example, if the distance traveled is 100000 inches, the program must display the message “1 mile(s), 1017 yard(s), 2 foot/feet, and 4 inch(es)”.

   It is given that ► 1 mile = 63360 inches ► 1 yard = 36 inches ► 1 foot = 12 inches

Chapter 14
Manipulating Strings

14.1 Introduction

Generally speaking, a string is anything that you can type using the keyboard, including letters, symbols (such as &, *, and @), and digits. In Java, a string is always enclosed in double quotes.

Below is a Java program that uses strings.

```
a = "Everything enclosed in double quotes is a string, even the numbers: "; b = "3, 54, 731"; System.out.println(a + b); System.out.println("You can even mix letters, symbols and digits like this: "); System.out.println("3 + 4 equals 7");
```

Many times programs deal with data that comes in the form of strings (text). Strings are everywhere—from word processors, to web browsers, to text messaging programs. Many exercises in this book actually make extensive use of strings. Even though Java supports many useful methods for manipulating strings, this chapter covers only those methods that are necessary for this book's purpose. However, if you need even more information you can visit one of the following addresses:
https://tinyurl.com/3575wcb8

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#method.summary

✎ *Java string methods (subprograms) can be used when there is a need to manipulate a string, for example, to isolate a number of characters from the string, remove spaces that might exist at the beginning of it, or convert all of its characters to uppercase.*

Methods are nothing more than small subprograms that solve small problems.

14.2 The Position of a Character in a String

Let's use the text “Hello World” in the following example. The string consists of 11 characters (including the space character between the two words). Java numerates characters assuming that the first one is at position 0, the second one is at position 1, and so on. The position of each character is shown here.

0	1	2	3	4	5	6	7	8	9	10
H	**e**	**l**	**l**	**o**		**W**	**o**	**r**	**l**	**d**

A space is a character just like any other character. Just because nobody can see it, it doesn't mean it doesn't exist!

14.3 Useful String Methods (Subprograms), and More

Trimming

Trimming is the process of removing whitespace characters from the beginning or the end of a string.

Some of the whitespace characters that are removed with the trimming process are: ► an ordinary space ► a tab ► a new line (line feed) ► a carriage return For example, you can trim any spaces that the user mistakenly entered at the end or at the beginning of a string.

The method that you can use to trim a string is

subject.**trim**()

This method returns a copy of `subject` in which any whitespace characters are removed from both the beginning and the end of the `subject` string.

Example

Project_14.3a

```
public static void main(String[] args) throws Exception
                                {
                        String a, b;
                a = " Hello "; b = a.trim();
   System.out.println(b + " Poseidon!"); //It displays:
                        Hello Poseidon!
```

```
    System.out.println(a + " Poseidon!"); //It displays:
                        Hello Poseidon!
                                }
```

✎ *Note that the content of variable* `a` *is not altered. If you do need to alter its content, you can use the statement* `a = a.trim();`

String replacement

subject.**replace**(*search, replace*)

This method searches in *subject* and returns a copy of it in which all occurrences of the *search* string are replaced with the *replace* string.

Example

```
                        🗀 Project_14.3b
public static void main(String[] args) throws Exception
                                {
                        String a, b;
            a = "I am newbie in C++. C++ rocks!"; b =
                        a.replace("C++", "Java");
    System.out.println(b); //It displays: I am newbie in
        Java. Java rocks System.out.println(a); //It
          displays: I am newbie in C++. C++ rocks }
```

✎ *Note that the content of variable* `a` *is not altered. If you do need to alter its content, you can use the statement* `a = a.replace("C++", "Java");`

Counting the number of characters

subject.**length**()

This method returns the length of *subject* or, in other words, the number of characters *subject* consists of.

Example

```
                        🗀 Project_14.3c
public static void main(String[] args) throws Exception
                                {
                        String a; int k;
                    a = "Hello Olympians!";
        System.out.println(a.length()); //It displays: 16
```

```
    k = a.length(); System.out.println(k); //It displays:
                                16
     System.out.println("I am newbie in Java".length());
                        //It displays: 19
                                }
```

A character includes anything you can type using the keyboard, such as letters, symbols, digits, and space characters.

Finding string position

subject.**indexOf**(*search*)

This method returns the numerical position of the first occurrence of `search` in `subject`, or `-1` if `search` is not found.

Example

```
                        Project_14.3d
public static void main(String[] args) throws Exception
                                {
                          int i; String a;
          a = "I am newbie in Java. Java rocks!"; i =
                          a.indexOf("newbie");
              System.out.println(i); //It displays: 5
    System.out.println(a.indexOf("Java")); //It displays:
                                15
     System.out.println(a.indexOf("C++")); //It displays:
                                -1
                                }
```

The first character is at position 0.

Converting to lowercase

subject.**toLowerCase**()

This method returns a copy of `subject` in which all the letters of the string `subject` are converted to lowercase.

Example

```
                        Project_14.3e
```

```
public static void main(String[] args) throws Exception
                                {
                            String a, b;
        a = "My NaMe is JohN"; b = a.toLowerCase();
   System.out.println(b); //It displays: my name is john
   System.out.println(a); //It displays: My NaMe is JohN
                                }
```

✎ *Note that the content of variable* `a` *is not altered. If you do need to alter its content, you can use the statement* `a = a.toLowerCase();`

Converting to uppercase

subject.**toUpperCase**()

This method returns a copy of `subject` in which all the letters of the string `subject` are converted to uppercase.

Example

```
                        🗀 Project_14.3f
public static void main(String[] args) throws Exception
                                {
                            String a, b;
        a = "My NaMe is JohN"; b = a.toUpperCase();
   System.out.println(b); //It displays: MY NAME IS JOHN
   System.out.println(a); //It displays: My NaMe is JohN
                                }
```

✎ *Note that the content of variable* `a` *is not altered. If you do need to alter its content, you can use the statement* `a = a.toUpperCase();`

Retrieving an individual character from a string

subject.**charAt**(*index*)

This method returns the character located at `subject`'s specified `index`. As already mentioned, the string indexes start from zero. You can use index 0 to access the first character, index 1 to access the second character, and so on. The index of the last character is 1 less than the length of the string.

Example

```
                        🗀 Project_14.3g
```

```
public static void main(String[] args) throws Exception
                                {
                            String a;
                    a = "Hello World";
    System.out.println(a.charAt(0)); //It displays the
                        first letter: H
     System.out.println(a.charAt(6)); //It displays: W
   System.out.println(a.charAt(10)); //It displays: d }
```

✎ *Note that the space between the words "Hello" and "World" is considered a character as well. So, the letter W exists in position 6 and not in position 5.*

If you attempt to use an invalid index such as a negative one or an index greater than the length of the string, Java throws an error message as shown in **Figure 14–1**.

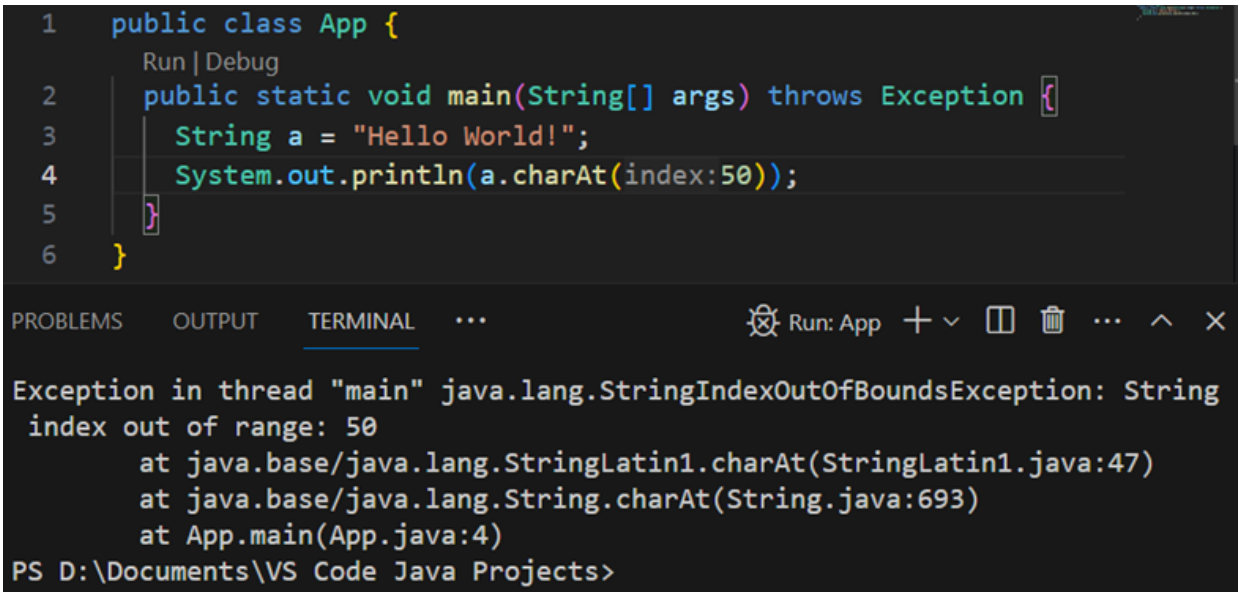


Figure 14–1 An error message indicating an invalid index

String indexes must be in a range from 0 to one less than the length of the string.

Getting part of a string

subject.**substring**(*beginIndex [, endIndex]*)

This method returns a portion of `subject`. Specifically, it returns the substring starting from position `beginIndex` and running up to, but not including,

position *endIndex*. The argument *endIndex* is optional. If it is omitted, the substring starting from position *beginIndex* until the end of *subject* is returned.

Example

```
Project_14.3h
public static void main(String[] args) throws Exception
{
String a;
a = "Hello Athena";
System.out.println(a.substring(6, 9)); //It displays:
Ath System.out.println(a.substring(7)); //It
displays: thena }
```

Comparing strings

subject.**compareTo**(*string*)

This method returns the value 0 when *subject* is lexicographically equal to *string*; a value less than 0 if *subject* is lexicographically less than *string*; and a value greater than 0 if *subject* is lexicographically greater than *string*.

The term “lexicographically” means that the letter “A” is considered “less than” the letter “B”, the letter “B” is considered “less than” the letter “C”, and so on. Of course, if two strings contain words in which the first letter is identical, Java moves on and compares their second letters and perhaps their third letters (if necessary). For example, the word “backspace” is considered “less than” the word “backwards” because the fifth letter, “s”, is “less than” the fifth letter, “w”.

Another more convenient method to check if two strings are equal is the following:

subject.**equals**(*string*)

This method checks if *subject* is lexicographically equal to *string* and returns `true` or `false` accordingly.

Example

```
Project_14.3i
public static void main(String[] args) throws Exception
{
String a = "backspace"; String b = "backwards";
String c = "Backspace"; String d = "winter"; String e
```

```
                    = "winter";
   System.out.println(a.compareTo(b)); //It displays: -4
   System.out.println(a.compareTo(c)); //It displays: 32
    System.out.println(a.compareTo(d)); //It displays:
                        -21
    System.out.println(d.compareTo(e)); //It displays: 0
   System.out.println(a.equals(d)); //It displays: false
   System.out.println(a.equals(c)); //It displays: false
    System.out.println(d.equals(e)); //It displays: true
                        }
```

✎ *Note that the letters “b” and “B” are considered two different letters.*

✎ *Consider the term 'lexicographically' in the context of how words are organized in an English dictionary.*

Converting a number or a character to a string

String.**valueOf**(*subject*)

This method returns a string version of `subject` or, in other words, it converts a number (real or integer) or character into a string.

Example

```
                    Project_14.3j
public static void main(String[] args) throws Exception
                        {
                String a; char b; int c, d;
                        b = 'W';
    a = String.valueOf(b); //Assign letter W to string
    variable a System.out.println(a); //It displays: W
                        c = 12;
                        d = 34;
       System.out.println(c + d); //It displays: 46
          System.out.println(String.valueOf(c) +
          String.valueOf(d)); //It displays: 1234
                        }
```

In Java, however, the same result can be achieved if you just start the sequence with a string, even an empty one, as shown in the example that follows.

Project_14.3k

```
public static void main(String[] args) throws Exception
                                {
                String a; char b; int c, d;
                            b = 'W';
    a = "" + b; //Assign letter W to string variable a
          System.out.println(a); //It displays: W
                            c = 12;
                            d = 34;
       System.out.println(c + d); //It displays: 46
    System.out.println("" + c + d); //It displays: 1234
       System.out.println("Result - " + c + d); //It
                    displays: Result - 1234
                                }
```

Exercise 14.3-1 Displaying a String Backwards

Write a Java program that prompts the user to enter any string with four letters and then displays its contents backwards. For example, if the string entered is "Zeus", the program must display "sueZ".

Solution

Let's say that user's input is assigned to variable `s`. You can access the fourth character using `s.charAt(3)`, the third character using `s.charAt(2)`, and so on.

The Java program is shown here. The concatenation operator (+) is used to reassemble the final reversed string.

Project_14.3-1

```
public static void main(String[] args) throws Exception {
   String s, sReversed;
   System.out.print("Enter a word with four letters: "); s = cin.nextLine();
   sReversed = "" + s.charAt(3) + s.charAt(2) + s.charAt(1) + s.charAt(0);
   System.out.println(sReversed); }
```

✎ *In Java, it is sometimes necessary to force the compiler to do concatenation, and not normal addition, by starting the sequence with a string, even an empty one.*

Exercise 14.3-2 Switching the Order of Names

Write a Java program that prompts the user to enter in one single string both first and last name. In the end, the program must change the order of the two names.

Solution

This exercise is not the same as the one that you learned in Exercises 8.1-3 and 8.1-4, which swapped the numeric values of two variables. In this exercise both the first and last names are entered in one single string, so the first thing that the program must do is split the string and assign each name to a different variable. If you manage to do so, then you can just rejoin them in a different order.

Let's try to understand this exercise using an example. The string that you must split and the position of its individual character are shown here.

0	1	2	3	4	5	6	7	8
T	**o**	**m**		**S**	**m**	**i**	**t**	**h**

The character that visually separates the first name from the last name is the space character between them. The problem here is that this character is not always at position 3. Someone can have a short first name like “Tom” and someone else can have a longer one like “Robert”. Thus, you need something that actually finds the position of the space character regardless of the content of the string.

Method `indexOf()` is what you are looking for! If you use it to find the position of the space character in the string “Tom Smith”, it returns the value 3. But if you use it to find the space character in another string, such as “Angelina Brown”, it returns the value 8 instead.

> ✎ *The value 3 is not just the position where the space character exists. It also represents the number of characters that the word “Tom” contains! The same applies to the value 8 that is returned for the string “Angelina Brown”. It represents both the position where the space character exists and the number of characters that the word “Angelina” contains!*

The Java program for this algorithm is shown here.

Project_14.3-2

```
public static void main(String[] args) throws Exception {
   String fullName, name1, name2; int spacePos;
   System.out.print("Enter your full name: "); fullName = cin.nextLine();
   //Find the position of space character. This is also the number //of characters
   first name contains spacePos = fullName.indexOf(" ");
   //Get spacePos number of characters starting from position 0
   name1 = fullName.substring(0, spacePos);
   //Get the rest of the characters starting from position spacePos + 1
   name2 = fullName.substring(spacePos + 1);
   fullName = name2 + " " + name1;
   System.out.println(fullName); }
```

*The method `subject`.substring(`beginIndex`, `endIndex`) returns a portion of `subject`. Specifically, it returns the substring starting from `beginIndex` position and running up to **but not** including `endIndex` position.*

Note that this program cannot be applied to a Spanish name such as "Maria Teresa García Ramírez de Arroyo". The reason is obvious!

Exercise 14.3-3 Creating a Login ID

Write a Java program that prompts the user to enter their last name and then creates a login ID from the first four letters of the name (in lowercase) and a three-digit random integer.

Solution

To create a random integer you can use the `Math.random()`method. Since you need a random integer of three digits, the range must be between 100 and 999.

The Java program for this algorithm is shown here.

Project_14.3-3

```
public static void main(String[] args) throws Exception {
   int randomInt; String lastName, loginID;
   System.out.print("Enter last name: "); lastName = cin.nextLine();
   //Get random integer between 100 and 999
   randomInt = 100 + (int)(Math.random() * 900);
   loginID = lastName.substring(0, 4).toLowerCase() + randomInt;
   System.out.println(loginID); }
```

✎ *Note that the method* `substring(0, 4)` *returns the substring starting from position 0 and running up to but not including position 4.*

✎ *Note how the method* `substring()` *is chained to the method* `toLowerCase()`. *The result of the first method is used as a* `subject` *for the second method. This is a writing style that most programmers prefer to follow because it helps to save a lot of code lines. Of course you can chain as many methods as you wish, but if you chain too many of them, no one will be able to understand your code.*

Exercise 14.3-4 Creating a Random Word

Write a Java program that displays a random word consisting of three letters.

Solution

To create a random word you need a string that contains all 26 letters of the English alphabet. Then you can use the `Math.random()` method to choose a random letter between position 0 and 25.

The Java program for this algorithm is shown here.

🗀 **Project_14.3-4a**

```
public static void main(String[] args) throws Exception
                                {
                String alphabet, randomWord;
          alphabet = "abcdefghijklmnopqrstuvwxyz";
   randomWord = "" + alphabet.charAt((int)(Math.random()
                          * 26)) +
                  alphabet.charAt((int)(Math.random() *
                                  26)) +
                  alphabet.charAt((int)(Math.random() *
                                  26));
             System.out.println(randomWord); }
```

🖻 *You can nest one subprogram within another. Note how the method* `random()` *is nested within the method* `charAt()`. *The result of the inner (nested) method is used as an argument for the outer method.*

✎ *Note that the method* `Math.random()` *is called three times and each time it may return a different random number.*

In Java, it is sometimes necessary to force the compiler to do concatenation, and not normal addition, by starting the sequence with a string, even an empty one.

You can also use the `length()` method to get the length of string `alphabet` as shown here.

```
                    Project_14.3-4b
public static void main(String[] args) throws Exception
                                {
                String alphabet, randomWord;
           alphabet = "abcdefghijklmnopqrstuvwxyz";
     randomWord = "" + alphabet.charAt((int)(Math.random()
                    * alphabet.length())) +
                    alphabet.charAt((int)(Math.random() *
                            alphabet.length())) +
                    alphabet.charAt((int)(Math.random() *
                             alphabet.length()));
              System.out.println(randomWord); }
```

Exercise 14.3-5 Finding the Sum of Digits

Write a Java program that prompts the user to enter a three-digit integer and then calculates the sum of its digits. Solve this exercise without using the integer remainder (%) operator.

Solution

Now you may wonder why this exercise is placed in this chapter, which primarily focuses on string manipulation. You might argue that you already know how to split a three-digit integer into its three digits and assign each digit to a separate variable as you did learn a method in Chapter 13 using the division (/) and the integer remainder (%) operators. So, why is this exercise discussed here again?

The reason is that Java is a very powerful language and you can use its magic forces to solve this exercise in a totally different way. The main idea is to convert the user-provided integer to type `String` and assign each digit (each character) into individual variables Project_14.3-5

```
public static void main(String[] args) throws Exception {
   int number, total; String sNumber, digit1, digit2, digit3;
```

```
System.out.print("Enter an three-digit integer: "); number =
Integer.parseInt(cin.nextLine());
sNumber = "" + number; //Convert number to string
digit1 = "" + sNumber.charAt(0); //Convert character at position 0 to string digit2
= "" + sNumber.charAt(1); //Convert character at position 1 to string digit3 = "" +
sNumber.charAt(2); //Convert character at position 2 to string
total = Integer.parseInt(digit1) + Integer.parseInt(digit2) +
Integer.parseInt(digit3);
System.out.println(total); }
```

✎ *As variables* `digit1`, `digit2`, *and* `digit3` *are of type* `String`, *you need to use the* `parseInt()` *method to convert them to integers before finding their sum in the variable* `total`.

14.4 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A string is anything that you can type using the keyboard.
2) Strings must be enclosed in parentheses.
3) The phrase “Hi there!” contains 8 characters.
4) In the phrase “Hi there!” the letter “t” is at position 3.
5) The statement `y = a.charAt(1)` assigns the second character of the string contained in variable `a` to variable `y`.
6) The following code fragment satisfies the property of definiteness.
   ```
   a = "Hello"; y = a.charAt(5);
   ```
7) Trimming is the process of removing whitespace characters from the beginning or the end of a string.
8) The statement `y = ("Hello Aphrodite").trim()` assigns the value “`HelloAphrodite`” to variable `y`.
9) The statement `System.out.println(("Hi there!").replace("Hi", "Hello"))` displays the message “Hello there!”.
10) The statement `index = ("Hi there").indexOf("the")` assigns the value 4 to variable `index`.
11) The statement `System.out.println(("hi there!").toUpperCase())` displays the message “`Hi There!`”.
12) The `statement System.out.println(("Hi there!").substring(0))` displays the message “`Hi there!`”

13) The statement `System.out.println(a.substring(0, a.length()))` displays some letters of the variable `a`.

14) The following statement is equivalent to the statement
`System.out.println(a.charAt(a.length() - 1));`
System.out.**println**(a.**substring**(a.**length**() - 1, a.**length**()));

15) The following statement displays the word "`HELLO`".

```
System.out.println(("hello there!").toUpperCase().substring(0, 5));
```

16) If variable `a` contains a string of 100 characters then the following statement is equivalent to the statement
`System.out.println(a.chartAt(99);`
System.out.**println**(a.**chartAt**(a.**length**() − 1));

17) The following code fragment displays the value of 23.

```
a = 2023;
System.out.println(String.valueOf(a).substring(2, 4));
```

14.5 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) Which of the following is **not** a string?
   a) "Hello there!"
   b) "13"
   c) "13.5"
   d) All of the above are strings.

2) In which position does the space character in the string "Hello Zeus!", exist?
   a) 6
   b) 5
   c) Space is not a character.
   d) none of the above

3) The statement
System.out.**println**(a.**substring**(a.**length**() - 2, a.**length**() - 1));
displays
   a) the last character of variable `a`.
   b) the second to last character of variable `a`.
   c) The statement is not valid.

4) The statement
y = a.**trim**().**replace**("*a*", "*b*").**replace**("*w*", "*y*");

is equivalent to the statement a) `y = a.replace("a", "b").replace("w", "y").trim();` b) `y = a.replace("a", "b").trim().replace("w", "y");` c) `y = a.trim().replace("w", "y").replace("a", "b");` d) all of the above 5) The statement `a.replace(" ", "")` a) adds a space between each letter in the variable `a`.

b) removes all space characters from the variable `a`.

c) empties the variable `a`.

6) The statement `(" Hello ").replace(" ", "")` is equivalent to the statement a) `(" Hello ").replace("", "")` b) `(" Hello ").trim()` c) all of the above d) none of the above 7) The following code fragment
a = "";
System.out.**println**(a.**length**());

displays

a) nothing.

b) 1.

c) 0.

d) The statement is invalid.

e) none of the above 8) Which value assigns the following code fragment to the variable `Shakespeare`?

```
toBeOrNotToBe = "2b or not 2b"; Shakespeare = toBeOrNotToBe.indexOf("b");
```

a) 1

b) 2

c) 11

d) none of the above 9) What does the following code fragment do?

```
a = "Hi there"; b = a.substring(a.indexOf(" ") + 1);
```

a) It assigns the word "`Hi`" to the variable `b`.

b) It assigns a space character to the variable `b`.

c) It assigns the word "`there`" to the variable `b`.

d) none of the above 10) The following code fragment
a = 15;

```
b = 5;
System.out.println(String.valueOf(a) + String.valueOf(b));
```

displays

a) 155.

b) 10.

c) 15 + 5

d) none of the above

14.6 Review Exercises

Complete the following exercises.

1) Write a Java program that creates and displays a random word consisting of five letters. The first letter must be a capital letter.
2) Write a Java program that prompts the user to enter their name and then creates a secret password consisting of three letters (in lowercase) randomly picked up from their name, and a random four-digit number. For example, if the user enters “Vassilis Bouras” a secret password can probably be one of “sar1359” or “vbs7281” or “bor1459”. Space characters are not allowed in the secret password.
3) Write a Java program that prompts the user to enter a three-digit integer and then reverses it. For example, if the user enters the number 375, the number 573 must be displayed. Solve this exercise without using the integer remainder (%) operator.
4) Write a Java program that prompts the user to enter their first name, middle name, and last name and displays them formatted in all the following ways.

   FirstName MiddleName LastName FirstName M. LastName (where M is the first letter of the middle name) LastName F. (where F is the first letter of the first name) Furthermore, the program must ensure that regardless of how the user enters their name, it will always be displayed with the first letter capitalized and the rest in lowercase.

   For example, assume that the user enters the following: First name: Aphrodite Middle name: MARIA Last name: boura The program must display the user's name formatted in all the following ways: Aphrodite Maria Boura Aphrodite M. Boura Boura A.

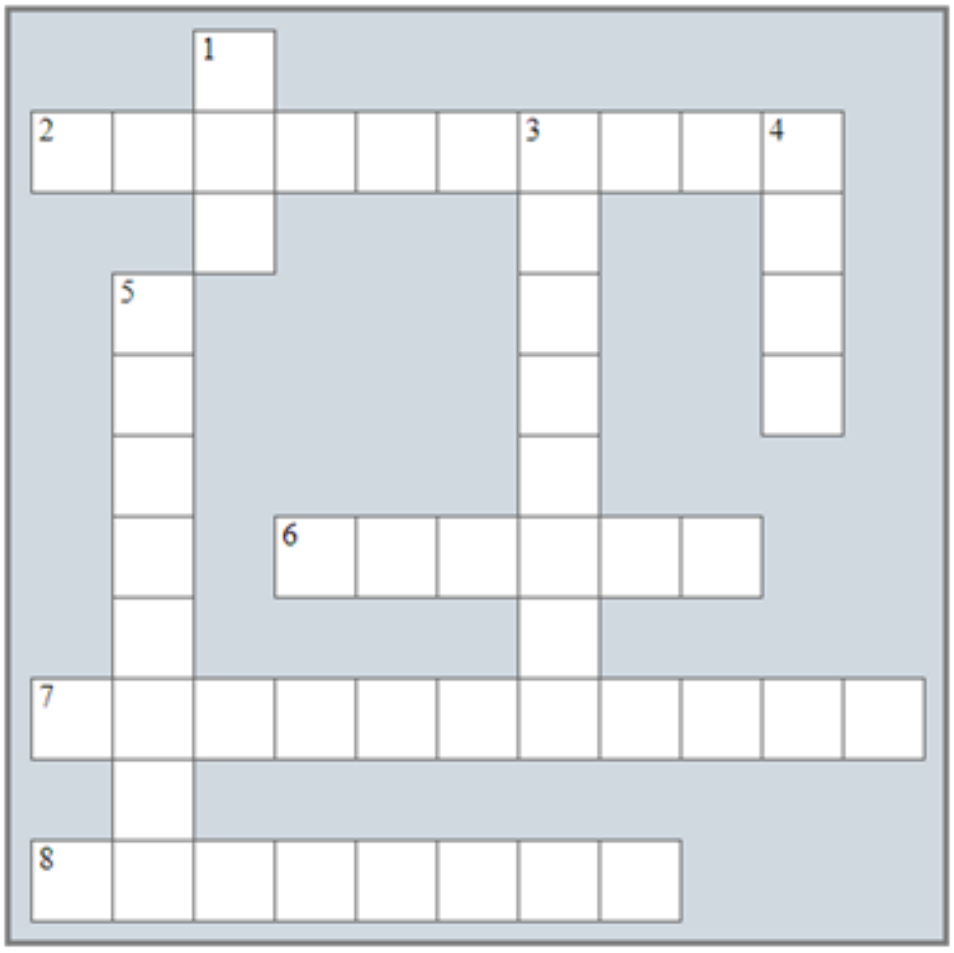
5) Some words such as “revolutionary” and “internationalization” are so lengthy that writing them out repeatedly can become quite tiresome. In such cases, these words can be replaced with a special abbreviation which is made like this: you keep the first and the last letter of a word and insert the number of letters between them. For instance, “revolutionary” becomes “r11y” and “internationalization” becomes “i18n”.

   Write a Java program that lets the user enter a long word and displays its abbreviation.

Review in “Sequence Control Structures”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

2) The `length()` method returns the number of _____ in a string.

6) Anything that you can type using the keyboard.

7) Java provides many ready-to-use _____.

8) This control structure refers to the line-by-line execution by which statements are executed sequentially.

Down

1) A whitespace character.

3) The process of removing whitespace characters from the beginning or the end of a string.

4) The `Math.sin()` method returns the _____ of a number.

5) The `Math.abs()` method returns the _____ value of a number.

Review Questions

Answer the following questions.

1) What is a sequence control structure?
2) What operations can a sequence control structure perform?
3) What does the term "type casting" mean in computer science?
4) Give some examples of how you can use the quotient and the remainder of an integer division.
5) What is a method?
6) What does the term “chain a method” mean?
7) What does the term “nest a method” mean?

Part IV
Decision Control Structures

Chapter 15 Making Questions

15.1 Introduction

All you have learned so far is the sequence control structure, where statements are executed sequentially, in the same order in which they appear in the program. However, in serious Java programming, rarely do you want the statements to be executed sequentially. Many times you want a block of statements to be executed in one situation and an entirely different block of statements to be executed in another situation.

15.2 What is a Boolean Expression?

Let's say that variable `x` contains a value of 5. This means that if you ask the question "*is* `x` *greater than 2?*" the answer is obviously "*Yes*". For a computer, these questions are called *Boolean expressions*. For example, if you write `x > 2`, this is a Boolean expression, and the computer must check whether or not the expression `x > 2` is `true` or `false`.

✎ *A Boolean expression is an expression that results in a Boolean value, that is, either* `true` *or* `false`*.*

✎ *Boolean expressions are questions and they should be read as "Is something equal to/greater than/less than something else?" and the answer is just a "Yes" or a "No"* (`true` *or* `false`*)* .

A decision control structure can evaluate a Boolean expression or a set of Boolean expressions and then decide which block of statements to execute.

15.3 How to Write Simple Boolean Expressions

A *simple Boolean expression* is written as *`Operand1`* ***`Comparison_Operator`*** *`Operand2`*

where

- *`Operand1`* and *`Operand2`* can be values, variables, constants, or mathematical expressions
- *`Comparison_Operator`* can be one of those shown in **Table 15-1**.

Comparison Operator	Description
==	Equal (not assignment)
!=	Not equal
>	Greater than
<	Less than

>=	Greater than or equal to
<=	Less than or equal to

Table 15-1 Comparison operators in Java Here are some examples of Boolean expressions: ► `x > y`. This Boolean expression is a question to the computer and can be read as "*is* `x` *greater than* `y`?"

- `x <= y`. This Boolean expression is also a question to the computer and can be read as "*is* `x` *less than or equal to* `y`?"
- `x != 3 * y + 4`. This can be read as "*is* `x` *not equal to the result of the expression* `3 * y + 4`?"
- `s.equals("Hello") == true`. This can be read as "*is* `s` *equal to the word 'Hello'?"* In other words, this question can be read as "*does* `s` *contain the word 'Hello'?*"

> *In Java, in order to test if two strings are lexicographically equal, you need to use the* `equals()` *method.*

- `x == 5` . This can be read as "*is* `x` *equal to 5?*"

> *A very common mistake that novice programmers make when writing Java programs is to confuse the value assignment operator with the equal operator. They frequently make the mistake of writing* `x = 5` *when they actually want to say* `x == 5`*.*

Exercise 15.3-1 Filling in the Table

Fill in the following table with the words "true" or "false" according to the values of the variables `a`*,* `b`*, and* `c`*.*

a	b	c	a == 10	b <= a	c > 3 * a − b
3	−5	7			
10	10	21			
−4	−2	−9			

Solution

The first two Boolean expressions are straightforward and need no further explanation.

Regarding the Boolean expression `c > 3 * a − b`, be careful with the cases where `b` is negative. For example, in the first line, `a` is equal to 3 and `b` is equal to −5. The result of the expression `3 * a - b` is `3 * 3 − (−5) = 3 * 3 + 5 = 14`.

Since the content of variable `c` (in the first line) is not greater than 14, the result of the Boolean expression `c > 3 * a − b` is `false`.

After a little work , the table becomes

a	b	c	a == 10	b <= a	c > 3 * a − b
3	**−5**	**7**	*false*	*true*	*false*
10	**10**	**21**	*true*	*true*	*true*
−4	**−2**	**−9**	*false*	*false*	*true*

15.4 Logical Operators and Complex Boolean Expressions

A *complex Boolean expression* can be built of simpler Boolean expressions and can be written as *BE1* ***Logical_Operator*** *BE2*

where

- *BE1* and *BE2* can be any Boolean expression.
- *Logical_Operator* can be one of those shown in **Table 15-2**.

Logical Operator	Description
&&	AND (also known as logical conjunction)
||	OR (also known as logical disjunction)
!	NOT (also known as negation or logical complement)

Table 15-2 Logical operators in Java

✎ When you combine simple Boolean expressions with logical operators, the whole Boolean expression is called a “complex Boolean expression”. For example, the expression `x == 3 && y > 5` is a complex Boolean expression.

✎ In flowcharts, this book uses the commonly accepted AND, OR, and NOT operators!

The AND (`&&`) operator When you use the AND (`&&`) operator between two Boolean expressions (*BE1* `&&` *BE2*), it means that the result of the whole complex Boolean expression is `true` only when both (*BE1* and *BE2*) Boolean expressions are `true`.

You can organize this information in something known as a *truth table*. A truth table shows the result of a logical operation between two or more Boolean expressions for all their possible combinations of values. The truth table for the AND (`&&`) operator is shown here.

BE1 (Boolean Expression 1)	*BE2* (Boolean Expression 2)	*BE1* && *BE2*

false	**false**	*false*
false	**true**	*false*
true	**false**	*false*
true	**true**	*true*

Are you still confused? You shouldn't be! It is quite simple! Let's see an example. The complex Boolean expression `name.equals("John") == true && age > 5` is `true` only when the variable `name` contains the word “John” (without the double quotes) **and** variable `age` contains a value greater than 5. Both Boolean expressions must be `true`. If at least one of them is `false`, for example, the variable `age` contains a value of 3, then the whole complex Boolean expression is `false`.

The OR (`||`) operator When you use the OR (`||`) operator between two Boolean expressions (*`BE1 || BE2`*), it means that the result of the whole complex Boolean expression is `true` when either the first (*`BE1`*) or the second (*`BE2`*) Boolean expression is `true` (at least one).

The truth table for the OR (`||`) operator is shown here.

`BE1` (Boolean Expression 1)	*`BE2`* (Boolean Expression 2)	*`BE1 || BE2`*
false	**false**	*false*
false	**true**	*true*
true	**false**	*true*
true	**true**	*true*

Let's see an example. The complex Boolean expression `name.equals("John") == true || name.equals("George") == true` is `true` when the variable `name` contains the word “John” **or** the word “George” (without the double quotes). At least one Boolean expression must be `true`. If both Boolean expressions are `false`, for example, the variable `name` contains the word “Maria”, then the whole complex Boolean expression is `false`.

The NOT (`!`) operator When you use the NOT (`!`) operator in front of a Boolean expression (`!(`*`BE`*`)`), it means that the result of the whole complex Boolean expression is `true` when the Boolean expression *`BE`* is `false` and vice versa.

The truth table for the NOT (`!`) operator is shown here.

BE (Boolean Expression)	! (*BE*)
false	*true*
true	*false*

For example, the complex Boolean expression `!(age > 5)` is `true` when the variable `age` contains a value less than or equal to 5. For instance, if the variable `age` contains a value of 6, then the whole complex Boolean expression is `false`.

✎ *The logical operator NOT (`!`) reverses the result of a Boolean expression. In Java, the Boolean expression must be enclosed in parentheses.*

Exercise 15.4-1 Calculating the Results of Complex Boolean Expressions

Calculate the results of the following complex Boolean expressions when variables `a`, `b`, `c`, *and* `d` *contain the values 5, 2, 7, and −3 respectively.*

i) `(3 * a + b / 47 - c * b / a > 23) && (b != 2)`

ii) `(a * b - c / 2 + 21 * c / 3) || (a >= 5)`

Solution

Don't be scared! The results can be found very easily. All you need is to recall what applies to AND (`&&`) and OR (`||`) operators.

i) The result of an AND (`&&`) operator is `true` when both Boolean expressions are `true`. If you take a closer look, the result of the Boolean expression on the right (`b != 2`) is `false`. So, you don't have to waste your time calculating the result of the Boolean expression on the left. The final result is definitely `false`.

ii) The result of an OR (`||`) operator is `true` when at least one Boolean expression is `true`. If you take a closer look, the result of the Boolean expression on the right (`a >= 5`) is actually `true`. So, don't bother calculating the result of the Boolean expression on the left. The final result is definitely `true`.

15.5 Assigning the Result of a Boolean Expression to a Variable

Given that a Boolean expression actually returns a value (`true` or `false`), this value can be directly assigned to a variable. For example, the statement

```
a = x > y;
```

assigns a value of `true` or `false` to Boolean variable `a`. It can be read as "*If the content of variable* `x` *is greater than the content of variable* `y`*, assign the value*

`true` *to variable* `a`*; otherwise, assign the value* `false`". This next example displays the value `true` on the screen.

Project_15.5

```
public static void main(String[] args) throws Exception {
   int x, y; boolean a;
   x = 8;
   y = 5;
   a = x > y;
   System.out.println(a); }
```

15.6 What is the Order of Precedence of Logical Operators?

A more complex Boolean expression may use several logical operators like the expression shown here `x > y || x == 5 && x <= z || !(z == 1)` So, a reasonable question is "which logical operation is performed first?"

Logical operators in Java follow the same precedence rules that apply to the majority of programming languages. The order of precedence is: logical complements (`!`) are performed first, logical conjunctions (`&&`) are performed next, and logical disjunctions (`||`) are performed at the end.

Higher Precedence ⇑ Lower Precedence	Logical Operator
	!
	&&
	||

Table 15-3 The order of precedence of logical operators

✎ You can always use parentheses to change the default precedence.

Exercise 15.6-1 Filling in the Truth Table

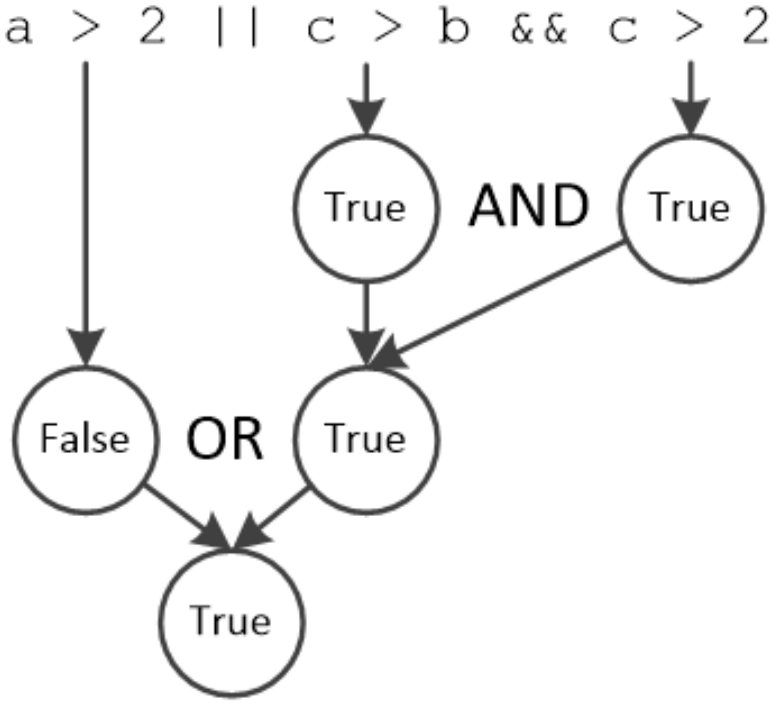
Fill in the following table with the words "true" or "false" according to the values of the variables `a`, `b` *and* `c`.

a	b	c	a > 2 || c > b && c > 2	!(a > 2 || c > b && c > 2)
1	−5	7		
10	10	3		
−4	−2	−9		

Solution

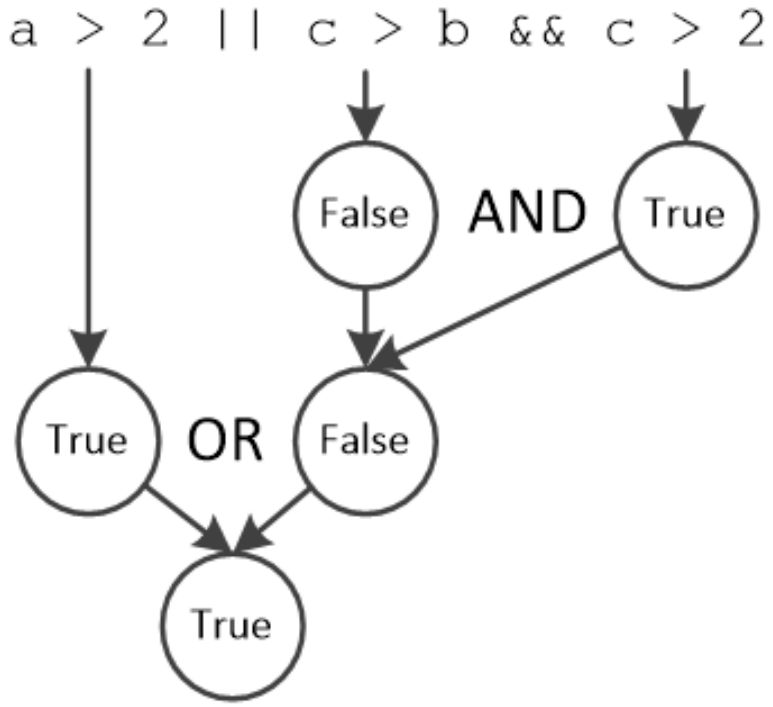
To calculate the result of complex Boolean expressions you can use the following graphical method.

For `a = 1, b = -5, c = 7`, the final result is `true` as shown here.

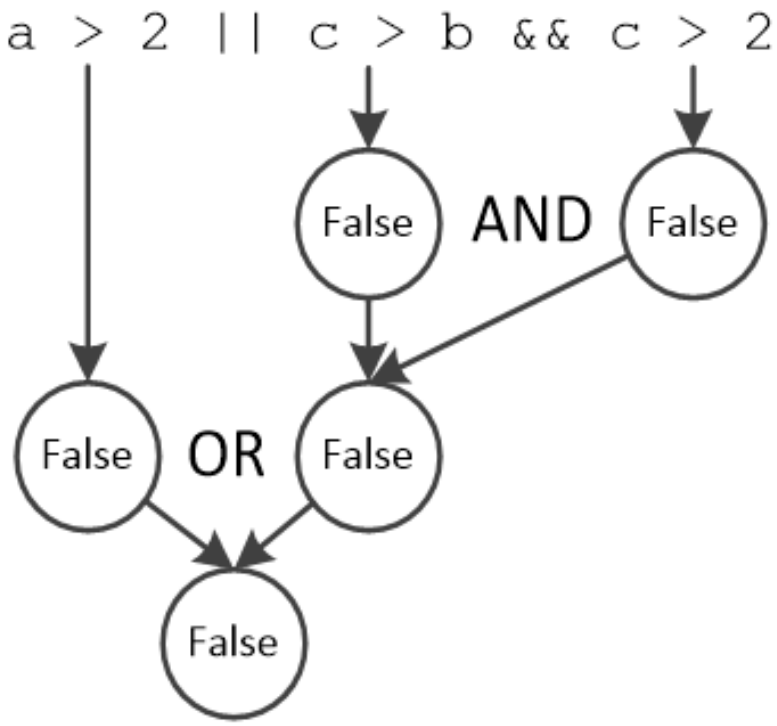


The AND (`&&`) operation has a higher precedence and is performed before the OR (`||`) operation.

For `a = 10, b = 10, c = 3`, the final result is `true` as shown here.



For `a = -4, b = -2, c = -9`, the final result is `false` as shown here.



The values in the table's fifth column can be calculated very easily because the Boolean expression in its column heading is almost identical to the one in the fourth column. The only difference is the NOT (!) operator in front of the expression. So, the values in the fifth column can be calculated by simply negating the results in the fourth column!

The final truth table is shown here.

a	b	c	a > 2 || c > b && c > 2	!(a > 2 || c > b && c > 2)
1	**−5**	**7**	*true*	*false*
10	**10**	**3**	*true*	*false*
−4	**−2**	**−9**	*false*	*true*

Exercise 15.6-2 Converting English Sentences to Boolean Expressions

A head teacher asks the students to raise their hands according to their age. He wants to find the students who are i) between the ages of 9 and 12.

ii) under the age of 8 and over the age of 11.

iii) 8, 10, and 12 years old.

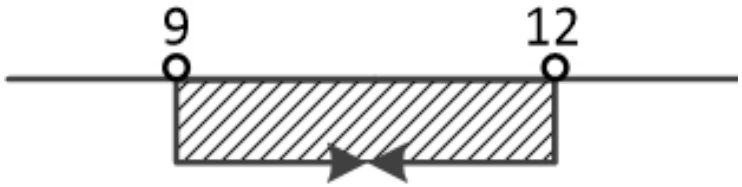
iv) between the ages of 6 and 8, and between the ages of 10 and 12.

v) neither 10 nor 12 years old.

Solution

To compose the required Boolean expressions, a variable `age` is used.

i) The sentence "*between the ages of 9 and 12*" can be graphically represented as shown here.



Be careful though! It is valid to write 9 ≤ age ≤ 12 in mathematics, but in Java the following is **not** possible `9 <= age <= 12`

What you can do is to split the expression into two parts, as shown here `age >= 9 && age <= 12`

> ✎ *For your confirmation, you can test this Boolean expression for several values inside and outside of the "region of interest" (the range of data that you have specified). For example, the result of the expression is* `false` *for the age values 7, 8, 13, and 17. On the contrary, for the age values 9, 10, 11, and 12, the result is* `true`*.*

ii) The sentence "*under the age of 8 and over the age of 11*" can be graphically represented as shown here.

> ✎ *Note the absence of the two circles that you saw in solution (i). This means the values 8 and 11 are* ***not*** *included within the two regions of interest.*

Be careful with the sentence "Under the age of 8 **and** over the age of 11". It's a trap! Don't make the mistake of writing `age < 8 && age > 11`

There is no person on the planet Earth that can be under the age of 8 **and** over the age of 11 concurrently!

The trap is in the word "**and**". Try to rephrase the sentence and make it *"Children! Please raise your hand if you are under the age of 8* ***or*** *over the age of 11"*. Now it's better and the correct Boolean expression becomes `age < 8 || age > 11`

✎ *For your confirmation, you can test this expression for several values inside and outside of the regions of interest. For example, the result of the expression is* `false` *for the age values 8, 9, 10 and 11. On the contrary, for the age values 6, 7, 12, and 15, the result is* `true`.

iii) Oops! Another trap in the sentence "*8, 10, and 12 years old*" with the "**and"** word again! Obviously, the next Boolean expression is wrong.

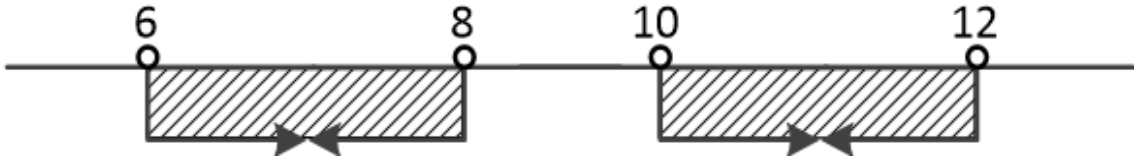
```
age == 8 && age == 10 && age == 12
```

As before, there isn't any student who is 8 **and** 10 **and** 12 years old concurrently! Once again, the correct Boolean expression must use the OR (`||`) operator.

```
age == 8 || age == 10 || age == 12
```

✎ *For your confirmation, you can test this expression for several values. For example, the result of the expression is* `false` *for the age values 7, 9, 11, and 13. For the age values 8, 10, and 12, the result is* `true`.

iv) The sentence "*between the ages of 6 and 8, and between the ages of 10 and 12*" can be graphically represented as shown here.



and the Boolean expression is `age >= 6 && age <= 8 || age >= 10 && age <= 12`

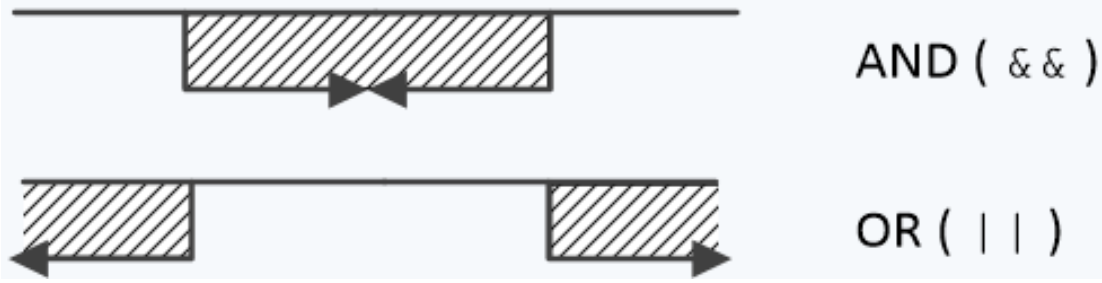
✎ *For your confirmation, the result of the expression is* `false` *for the age values 5, 9, 13, and 16. For the age values 6, 7, 8, 10, 11, and 12, the result is* `true`.

v) The Boolean expression for the sentence "*neither 10 nor 12 years old*" can be written as `age != 10 && age != 12`

or as

```
!(age == 10 || age == 12)
```

✎ When the arrows of the region of interest point towards each other, use the logical operator AND (`&&`). Otherwise, use OR (`||`) when the arrows point in opposite directions.



15.7 What is the Order of Precedence of Arithmetic, Comparison, and Logical Operators?

In many cases, an expression may contain different type of operators, such as the one shown here.

```
a * b + 2 > 21 || !(c == b / 2) && c > 13
```

In such cases, arithmetic operations are performed first, comparison operations are performed next, and logical operations are performed at the end, as shown in the following table.

Higher Precedence **Lower Precedence**	Arithmetic Operators	*, /, %
		+, –
	Comparison Operators	<, <=, >, >=, ==, !=
	Logical Operators	!
		&&
		||

Table 15-4 The order of precedence of arithmetic, comparison, and logical operators

15.8 How to Negate Boolean Expressions

Negation is the process of reversing the meaning of a Boolean expression. There are two approaches used to negate a Boolean expression.

First approach

The first approach is the easiest one. Just use a NOT (`!`) operator in front of the original Boolean expression and your negated Boolean expression is ready! For example, if the original Boolean expression is `x > 5 && y == 3`

the negated Boolean expression becomes `!(x > 5 && y == 3)`

✎ Note that the entire expression must be enclosed in parentheses. It would be completely incorrect if you had written the expression as `!(x > 5) && y == 3`. In this case the NOT (`!`) operator would negate only the first Boolean expression, `x > 5`.

Second approach

The second approach is a little bit more complex but not difficult to learn. All you must do is negate every operator according to the following table.

Original Operator	Negated Operator
==	!=
!=	==
>	<=
<	>=
<=	>
>=	<
&&	||
||	&&
!	!

> ✎ *Note that the NOT (`!`) operator remains intact.*

For example, if the original Boolean expression is `x > 5 && y == 3`

the negated Boolean expression becomes `x <= 5 || y != 3`

However, there is a small detail that you should be careful with. If both AND (`&&`) and OR (`||`) operators co-exist in a complex Boolean expression, then the expressions that use the OR (`||`) operators in the negated Boolean expression must be enclosed in parentheses, in order to preserve the initial order of precedence. For example, if the original Boolean expression is `x >= 5 && x <= 10 || y == 3`

the negated Boolean expression must be `(x < 5 || x > 10) && y != 3`

> ✎ *If you forget to enclose the expression `x < 5 || x > 10` in parentheses, since the AND (`&&`) operator has a higher precedence than the OR (`||`) operator, the expression `x > 10 && y != 3` is evaluated first, which is wrong of course!*

Exercise 15.8-1 Negating Boolean Expressions

Negate the following Boolean expressions using both approaches.

i) `b != 4`

ii) `a * 3 + 2 > 0`

iii) `!(a == 5 && b >= 7)`

iv) `a == true`

v) `b > 7 && !(x > 4)`

vi) `a == 4 || b != 2`

Solution

First approach

i) `!(b != 4)` ii) `!(a * 3 + 2 > 0)` iii) `!(!(a == 5 && b >= 7))`, or the equivalent `a == 5 && b >= 7`

> ✎ *Two negations result in an affirmative. That is, two NOT (`!`) operators in a row negate each other.*

iv) `!(a == true)` v) `!(b > 7 && !(x > 4))` vi) `!(a == 4 || b != 2)` Second approach

i) `b == 4`

ii) `a * 3 + 2 <= 0`

> ✎ *Note that arithmetic operators are **not** "negated". Never substitute, for example, the plus (+) with a minus (−) operator!*

iii) `!(a != 5 || b < 7)`

✎ Note that the NOT (`!`) operator remains intact.

iv) `a != true` v) `b <= 7 || !(x <= 4)` vi) `a != 4 && b == 2`

15.9 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A Boolean expression is an expression that always results in one of two values.
2) A Boolean expression includes at least one logical operator.
3) In Java, the expression `x = 5` tests if the variable `x` is equal to 5.
4) The following statement is not a valid Java statement.

```
a = b == c;
```

5) The Boolean expression `b < 5` tests if the variable `b` is 5 or less.
6) The AND (`&&`) operator is also known as a logical disjunction operator.
7) The OR (`||`) operator is also known as a logical complement operator.

8) The result of a logical conjunction of two Boolean expressions equals the result of the logical disjunction of them, given that both Boolean expressions are `true`.

9) The result of a logical disjunction of two Boolean expressions is definitely `true`, given that the Boolean expressions have different values.

10) The expression `c == 3 && d > 7` is considered a complex Boolean expression.

11) The result of the logical operator OR (`||`) is `true` when both operands (Boolean expressions) are `true`.

12) The result of the Boolean expression `!(x == 5)` is `true` when the variable `x` contains any value except 5.

13) The NOT (`!`) operator has the highest precedence among logical operators.

14) The OR (`||`) operator has the lowest precedence among logical operators.

15) In the Boolean expression `(x > y || x == 5) && x <= z`, the AND (`&&`) operation is performed before the OR (`||`) operation.

16) In the Boolean expression `a * b + c > 21 || c == b / 2`, the program first tests if `c` is greater than 21.

17) When a teacher wants to find the students who are under the age of 8 and over the age of 11, the corresponding Boolean expression is `age < 8 && age > 11`.

18) The Boolean expression `x < 0 && x > 100` is, for any value of `x`, always `false`.

19) The Boolean expression `x > 0 || x < 100` is, for any value of `x`, always `true`.

20) The Boolean expression `x > 5` is equivalent to `!(x < 5)`.

21) The Boolean expression `!(x > 5 && y == 5)` is not equivalent to `!(x > 5) && y == 5`.

22) In William Shakespeare[13]'s *Hamlet* (Act 3, Scene 1), the main character says “To be, or not to be: that is the question:…. ” If you write this down as a Boolean expression `toBe || !toBe`, the result of this “Shakesboolean” expression is `true` for the following code fragment.

```
toBe = 1 > 0;
thatIsTheQuestion = toBe || !toBe;
```

23) The Boolean expression `!(!(x > 5))` is equivalent to `x > 5`.

15.10 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) Which of the following is **not** a comparison operator?
   a) >=
   b) ==
   c) <
   d) All of the above are comparison operators.
2) Which of the following is not a Java logical operator?
   a) &&
   b) ||
   c) All of the above are logical operators.
   d) None of the above is a logical operator.
3) If variable `x` contains a value of 5, what value does the statement `y = x % 2 == 1` assign to variable `y`?
   a) `true`
   b) `false`
   c) 1
   d) none of the above
4) If variable `x` contains a value of 5, what value does the statement `y = x % 2 == 0 || (int)(x / 2.0) == 2` assign to variable `y`?
   a) `true`
   b) `false`
   c) none of the above
5) The temperature in a laboratory room must be between 50 and 80 degrees Fahrenheit. Which of the following Boolean expressions tests for this condition?
   a) `t >= 50 || t <= 80`
   b) `50 >= t && t >= 80`
   c) `t >= 50 && t <= 80`
   d) `t > 50 || t < 80`
   e) none of the above
6) Which of the following is equivalent to the Boolean expression `t == 3 || t > 30`?
   a) `t == 3 && !(t <= 30)`
   b) `t == 3 && !(t < 30)`
   c) `!(t != 3) || !(t < 30)`
   d) `!(t != 3 && t <= 30)`
   e) none of the above

15.11 Review Exercises

Complete the following exercises.

1) Match each element from the first column with one or more elements from the second column.

Description	Operator
i) Logical operator	a) `%`

ii) Arithmetic operator	b) `+=`
iii) Comparison operator	c) `&&`
iv) Assignment operator (in general)	d) `==`
	e) `||`
	f) `>=`
	g) `!`
	h) `=`
	i) `*=`
	j) `/`

2) Fill in the following table with the words “true” or “false” according to the values of variables `a`, `b`, and `c`.

a	b	c	a != 1	b > a	c / 2 > 2 * a
3	**−5**	**8**			
1	**10**	**20**			
−4	**−2**	**−9**			

3) Fill in the following table with the words “true” or “false” according to the values of the Boolean expressions BE1 and BE2.

BE1 (Boolean Expression 1)	BE2 (Boolean Expression 2)	BE1 || BE2	BE1 && BE2	!(BE2)
false	**false**			
false	**true**			
true	**false**			
true	**true**			

4) Fill in the following table with the words “true” or “false” according to the values of variables `a`, `b`, and `c`.

a	b	c	a > 3 || c > b && c > 1	a > 3 && c > b || c > 1
4	**−6**	**2**		

−3	**2**	**−4**		
2	**5**	**5**		

5) For `x = 4`, `y = −2` and `flag = true`, fill in the following table with the corresponding values.

Expression	Value
`Math.pow(x + y, 3)`	
`(x + y) / (Math.pow(x, 2) − 14)`	
`x − 1 == y + 5`	
`x > 2 && y == 1`	
`x == 1 || y == −2 && !(flag == false)`	
`!(x >= 3) && (x % 2 > 1)`	

6) Calculate the result of each the following complex Boolean expressions when variables `a`, `b`, `c`, and `d` contain the values 6, −3, 4, and 7 respectively.

i) `(3 * a + b / 5 - c * b / a > 4) && (b != 3)` ii) `(a * b - c / 2 + 21 * c / 3 != 8) || (a >= 5)` Hint: Start by evaluating the simpler parts of the expressions.

7) A head teacher asks the students to raise their hands according to their age. He wants to find the students who are:

i) under the age of 12, but not those who are 8 years old.

ii) between the ages of 6 and 9, and also those who are 11 years old.

iii) over the age of 7, but not those who are 10 or 12 years old.

iv) 6, 9, and 11 years old.

v) between the ages of 6 and 12, but not those who are 8 years old.

vi) neither 7 nor 10 years old.

To compose the required Boolean expressions, use a variable named `age`.

8) Negate the following Boolean expressions without adding the NOT (`!`) operator in front of the expressions.

i) `x == 4 && y != 3`

ii) `x + 4 <= 0`

iii) `!(x > 5) || y == 4`

iv) `x != false`

v) `!(x >= 4 || z > 4)`

vi) `x != 2 && x >= -5`

9) As you already know, two negations result in an affirmative. Write the equivalent of the following Boolean expressions by negating them twice (applying both methods).

i) `x >= 4 && y != 10`

ii) `x - 2 >= 9`

iii) `!(x >= 2) || y != 4`

iv) `x != false || y == 3`

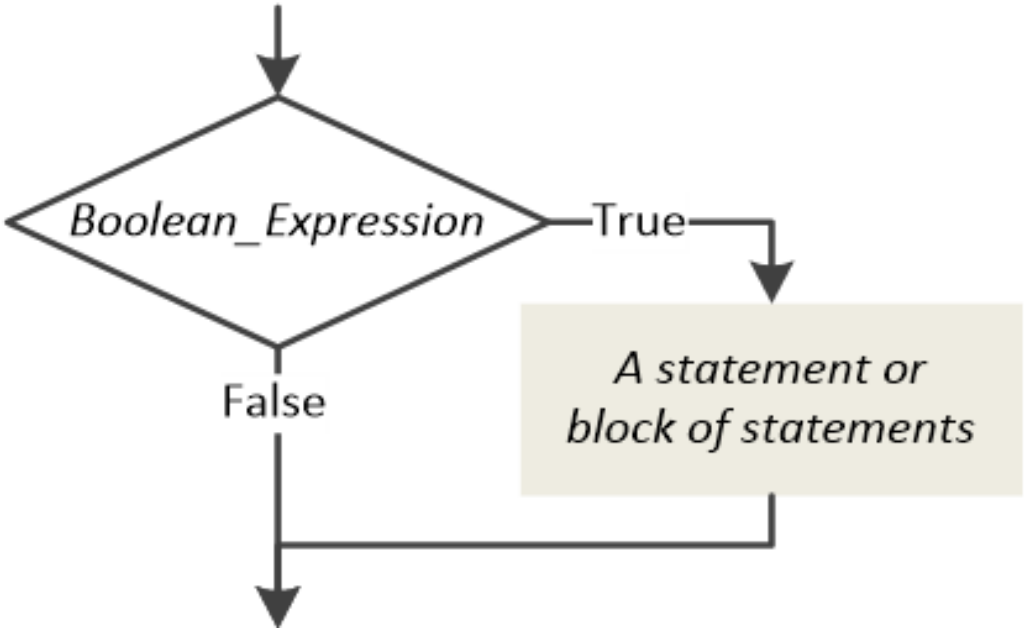
v) `!(x >= 2 && y >= 2)`

vi) `x != -2 && x <= 2`

Chapter 16
The Single-Alternative Decision Structure

16.1 The Single-Alternative Decision Structure

This is the simplest decision control structure. It includes a statement or block of statements on the "true" path only, as presented in the following flowchart fragment, given in general form.



If *Boolean_Expression* evaluates to `true`, the statement, or block of statements, of the structure is executed; otherwise, the statements are skipped.

The general form of the Java statement is

if (*Boolean_Expression*) {

```
    A statement or block of statements
```

}

> ✎ *Note that the statement or block of statements is indented by 2 extra spaces.*

In the next example, the message "You are underage!" displays only when the user enters a value less than 18. Nothing is displayed when the user enters a value that is greater than or equal to 18.

```
                    🗀 Project_16.1a
public static void main(String[] args) throws Exception
                              {
                          int age;
        System.out.print("Enter your age: "); age =
             Integer.parseInt(cin.nextLine());
                       if (age < 18) {
           System.out.println("You are underage!");
                              }
                             }
```

✎ *Note that the* `System.out.println()` *statement is indented by 2 + 2 = 4 spaces.*

In the next example, the message “You are underage!” and the message “You have to wait for a few more years” are displayed only when the user enters a value less than 18. Same as previously, no messages are displayed when the user enters a value that is greater than or equal to 18.

```
                    🗀 Project_16.1b
public static void main(String[] args) throws Exception
                              {
                          int age;
        System.out.print("Enter your age: "); age =
             Integer.parseInt(cin.nextLine());
                       if (age < 18) {
           System.out.println("You are underage!");
        System.out.println("You have to wait for a few
                        more years.");
                              }
                             }
```

✎ *All statements that appear inside an* `if` *statement should be indented to the right by the same number of spaces. In the previous example, both* `System.out.println()` *statements are indented by 2 spaces.*

✎ *To save paper, this book uses 2 spaces per indentation level. Java's official website, however, recommends the use of 4 spaces per indentation level.*

✎ *In order to indent the text cursor, instead of typing space characters, you can hit the "Tab ⇆" key once!*

✎ *In order to indent an existing statement or a block of statements, select it and hit the "Tab ⇆" key!*

✎ *In order to unindent a statement or a block of statements, select it and hit the "Shift ↑ + Tab ⇆" key combination!*

In the next example, the message "You are the King of the Gods!" is displayed only when the user enters the name "Zeus". However, the message "You live on Mount Olympus" is always displayed, no matter what name the user enters.

```
                    Project_16.1c
public static void main(String[] args) throws Exception
                                {
                          String name;
   System.out.print("Enter the name of an Olympian: ");
                       name = cin.nextLine();
                 if (name.equals("Zeus") == true) {
            System.out.println("You are the King of the
                                Gods!");
                                }
    System.out.println("You live on Mount Olympus."); }
```

✎ *Note that the last* `System.out.println()` *statement does* ***not*** *belong to the block of statements of the single-alternative decision structure.*

✎ *A very common mistake that novice programmers make when writing Java programs is to confuse the value assignment operator with the "equal" operator. They frequently make the mistake of writing* `if (name.equals("Zeus") = true)` *when they actually want to say* `if (name.equals("Zeus") == true)`.

When only one single statement is enclosed in the `if` statement, you can omit the braces { }. Thus, the `if` statement can be written as follows

if (*Boolean_Expression*) *One_Single_Statement*;

or you can even write the whole decision structure on one single line, like this:

if (*Boolean_Expression*) *One_Single_Statement*;

The braces are required only when more than one statement is enclosed within an `if` statement. Without the braces, Java assumes that only the next statement in order is part of the decision control structure.

However, to prevent potential logic errors, many programmers prefer to always use braces, even if the `if` statement includes only one single statement. In both of the following examples the `System.out.print(x)` statement is **not** part of the `if` statement.

```
if (x == y) x++;
   System.out.print(x);
if (x == y) {
   x++;
}
System.out.print(x);
```

Exercise 16.1-1 Trace Tables and Single-Alternative Decision Structures

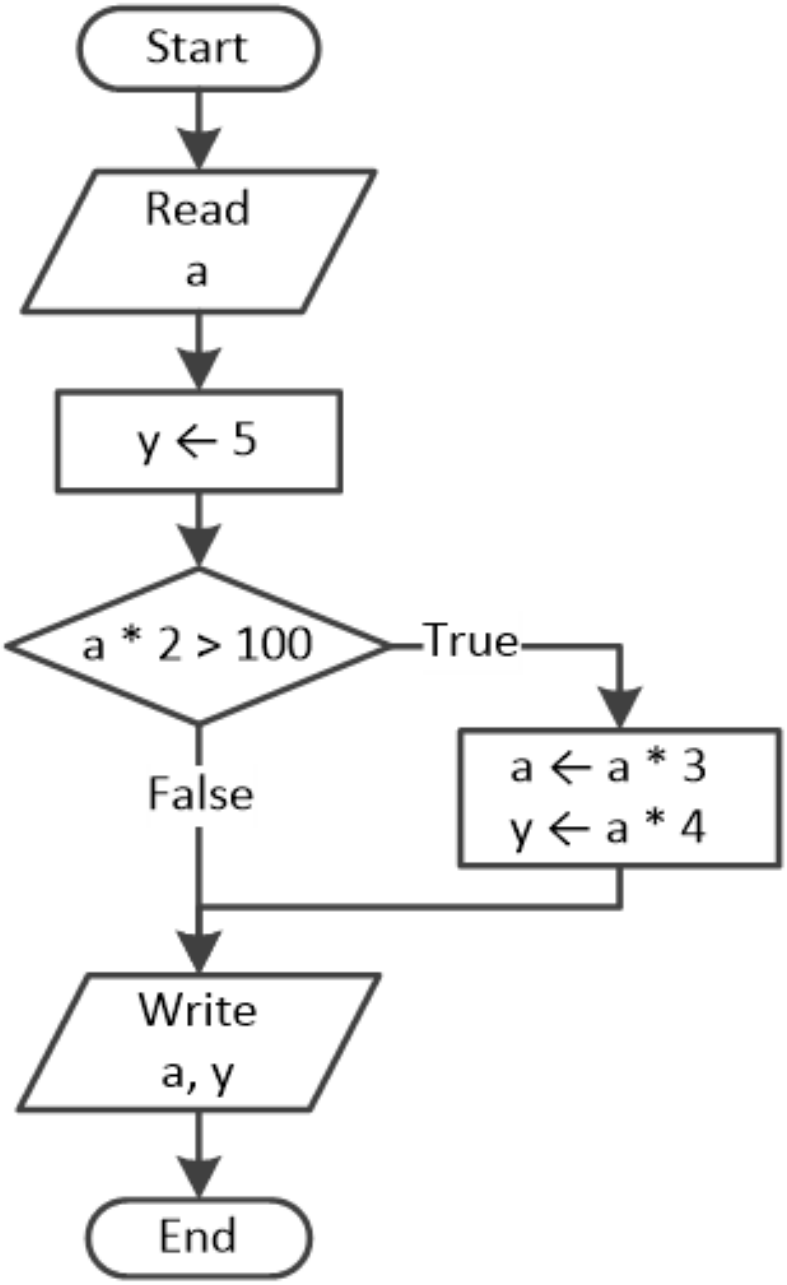
Design the corresponding flowchart and create a trace table to determine the values of the variables in each step of the next Java program for two different executions.

The input values for the two executions are: (i) 10, and (ii) 51.

Project_16.1-1

```
public static void main(String[] args) throws Exception {
   int a, y;
   a = Integer.parseInt(cin.nextLine());
   y = 5;
   if (a * 2 > 100) {
      a = a * 3;
      y = a * 4;
   }
   System.out.println(a + " " + y); }
```

Solution The flowchart is shown here.



The trace tables for each input are shown here.

i) For the input value of 10, the trace table looks like this.

Step	Statement	Notes	a	y
1	`a = Integer.parseI…`	User enters the value 10	**10**	?
2	`y = 5`		10	**5**
3	`if (a * 2 > 100)`	This evaluates to `false`		

4	`.println(a + " " + y)`	It displays: 10 5

ii) For the input value of 51, the trace table looks like this.

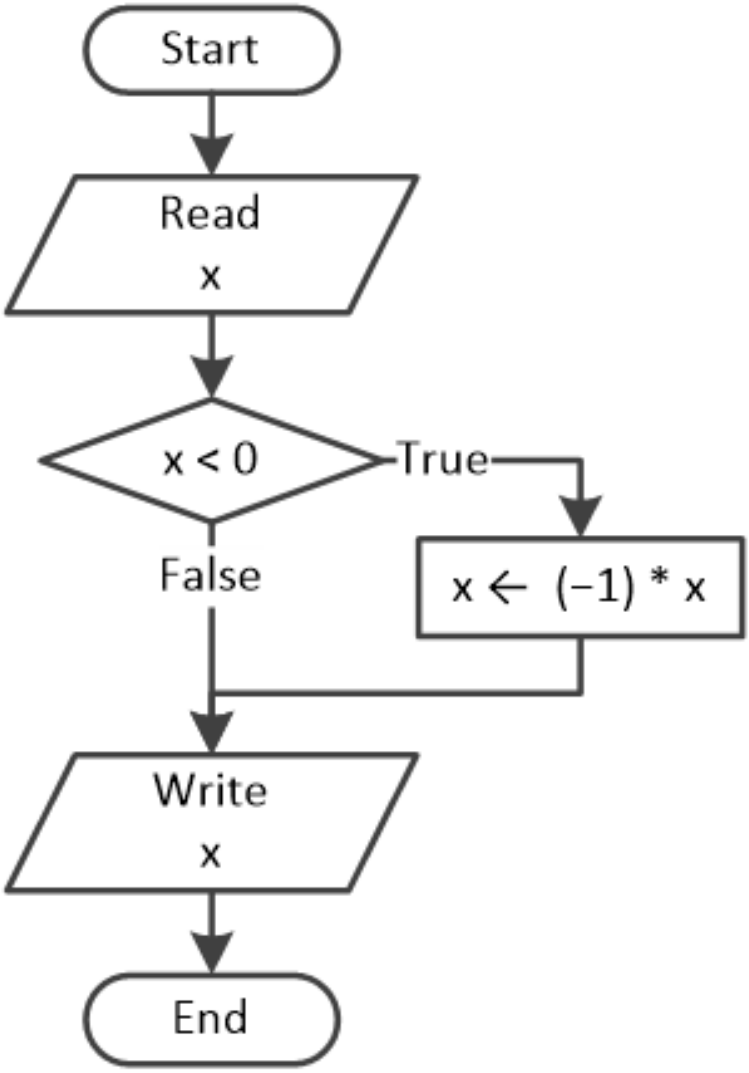
Step	Statement	Notes	a	y
1	`a = Integer.parseI…`	User enters the value 51	**51**	?
2	`y = 5`		51	**5**
3	`if (a * 2 > 100)`	This evaluates to `true`		
4	`a = a * 3`		**153**	5
5	`y = a * 4`		153	**612**
6	`.println(a + " " + y)`	It displays: 153 612		

Exercise 16.1-2 The Absolute Value of a Number

Design a flowchart and write the corresponding Java program that lets the user enter a number and then displays its absolute value.

Solution Actually, there are two approaches. The first approach uses a single-alternative decision structure, whereas the second one uses the built-in `Math.abs()` method.

First approach – Using a single-alternative decision structure The approach is simple. If the user enters a negative value, for example −5, this value is changed and displayed as +5. A positive number or zero, however, remains as is. The solution is shown in the flowchart that follows.



The corresponding Java program is as follows.

```
Project_16.1-2a
public static void main(String[] args) throws Exception
                                {
                              double x;
                  x = Double.parseDouble(cin.nextLine());
                             if (x < 0) {
                              x = (-1) * x;
                                  }
                          System.out.println(x); }
```

Second approach – Using the `Math.abs()` method In this case, you need just a few lines of code without any decision control structure!

```
Project_16.1-2b
public static void main(String[] args) throws Exception
                                {
                          double x;
              x = Double.parseDouble(cin.nextLine());
                 System.out.println(Math.abs(x)); }
```

16.2 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) The single-alternative decision structure is used when a sequence of statements must be executed.
2) You use a single-alternative decision structure to allow other programmers to more easily understand your program.
3) It is a possible that none of the statements enclosed in a single-alternative decision structure will be executed.
4) In a flowchart, the Decision symbol represents the beginning and the end of an algorithm.
5) The following code is syntactically correct.

```
static final int if = 5; public static void main(String[] args) throws
Exception {
   int x; x = if + 5; System.out.println(x); }
```

6) The single-alternative decision structure uses the reserved keyword `else`.
7) The following code fragment satisfies the property of definiteness.

```
if (b != 3) {
   x = a / (b - 3); }
```

8) The following Java program satisfies the property of definiteness.

```
public static void main(String[] args) throws Exception {
   double a, b, x;
   a = Double.parseDouble(cin.nextLine()); b =
   Double.parseDouble(cin.nextLine());
   if (b != 3) {
      x = a / (b - 3);
   }
   System.out.println(x); }
```

16.3 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The single-alternative decision structure is used when
   a) statements are executed one after another.
   b) a decision must be made before executing some statements.
   c) none of the above
   d) all of the above

2) The single-alternative decision structure includes a statement or block of statements on
   a) the false path only.
   b) both paths.
   c) the true path only.

3) In the following code fragment,

```
if (x == 3) x = 5;
    y++;
```

the statement y++ is executed
   a) only when variable x contains a value of 3.
   b) only when variable x contains a value of 5.
   c) only when variable x contains a value other than 3.
   d) always.

4) In the following code fragment,

```
if (x % 2 == 0) y++;
```

the statement y++ is executed when
   a) variable x is exactly divisible by 2.
   b) variable x contains an even number.
   c) variable x does not contain an odd number.
   d) all of the above
   e) none of the above

5) In the following code fragment,

```
x = 3 * y;
if (x > y) y++;
```

the statement y++ is
   a) always executed.
   b) never executed.
   c) executed only when variable y contains positive values.
   d) none of the above

16.4 Review Exercises

Complete the following exercises.

1) Identify the syntax errors in the following Java program:

```
public static void main(String[] args) throws Exception {
    double x, y
    x = Double.parseDouble(cin.nextLine();
    y ← -5; if (x * y / 2 > 20)
    y =* 2
        x += 4 * x²;
    }
    System.out.println(x y); }
```

2) Create a trace table to determine the values of the variables in each step of the following Java program for two different executions. Then, design the corresponding flowchart.

   The input values for the two executions are: (i) 10, and (ii) −10.

```
public static void main(String[] args) throws Exception {
   double x, y;
   x = Double.parseDouble(cin.nextLine());
   y = -5; if (x * y / 2 > 20) {
      y--;
      x -= 4;
   }
   if (x > 0) {
      y += 30;
      x = Math.pow(x, 2);
   }
   System.out.println(x + ", " + y); }
```

3) Create a trace table to determine the values of the variables in each step of the following Java program for two different executions. Then, design the corresponding flowchart.

   The input values for the two executions are: (i) −11, and (ii) 11.

```
public static void main(String[] args) throws Exception {
   int x, y;
   x = Integer.parseInt(cin.nextLine());
   y = 8;
   if (Math.abs(x) > 10) {
      y += x;
      x--;
```

```
   }
   if (Math.abs(x) > 10) {
      y *= 3;
   }
   System.out.println(x + ", " + y); }
```

4) Create a trace table to determine the values of the variables in each step of the following Java program for two different executions. Then, design the corresponding flowchart.

   The input values for the two executions are: (i) 1, 2, 3; and (ii) 4, 2, 1.

```
public static void main(String[] args) throws Exception {
   int x, y, z;
   x = Integer.parseInt(cin.nextLine()); y = Integer.parseInt(cin.nextLine());
   z = Integer.parseInt(cin.nextLine());
   if (x + y > z)
   x = y + z;
   if (x > y + z)
   y = x + z;
   if (x > y - z)
   z = x - z % 2;
   System.out.println(x + ", " + y + ", " + z); }
```

5) Write a Java program that prompts the user to enter a number, and then displays the message “Positive” when the user-provided number is positive.

6) Write a Java program that prompts the user to enter two numbers, and then displays the message “Both Positives” when both user-provided numbers are positives.

7) Write a Java program that prompts the user to enter their age and then displays the message “You can drive a car in Kansas (USA)” when the user-provided age is greater than 14.

8) Write a Java program that prompts the user to enter a string, and then displays the message “Uppercase” when the user-provided string contains only uppercase characters.

   Hint: Use the `equals()` and `toUpperCase()` methods.

9) Write a Java program that prompts the user to enter a string, and then displays the message “Many characters” when the user-provided string contains more than 20 characters.

   Hint: Use the `length()` method.

10) Write a Java program that prompts the user to enter four numbers and, if at least one of them is negative, it displays the message “Among the provided numbers, there is a negative one!”

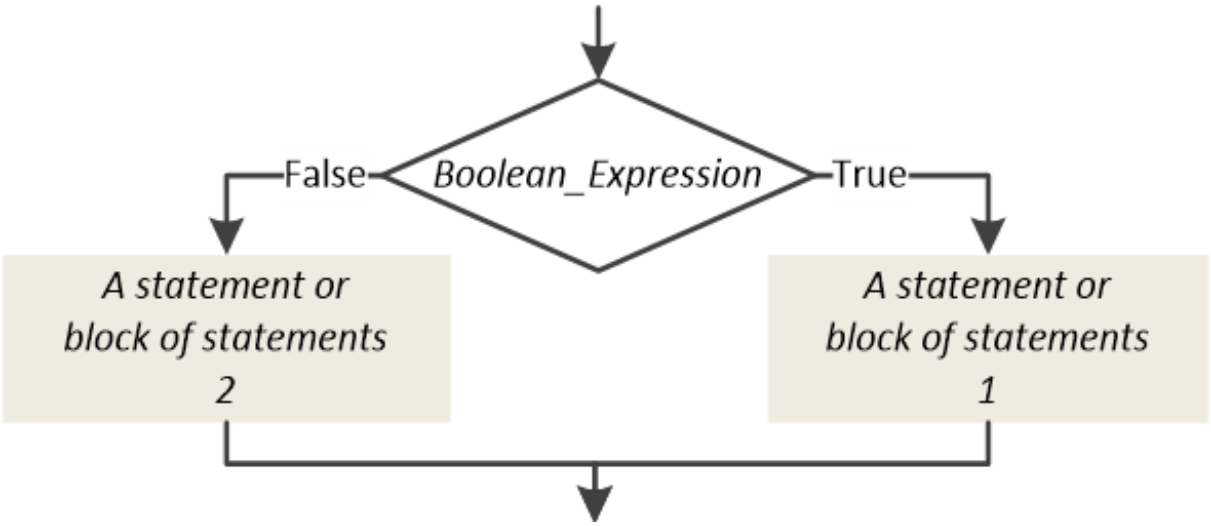
11) Write a Java program that prompts the user to enter two numbers. If the first user-provided number is greater than the second one, the program must swap their values. In the end, the program must display the numbers, always in ascending order.

12) Write a Java program that prompts the user to enter three temperature values measured at three different points in New York, and then displays the message “Heat Wave” if the average value is greater than 60 degrees Fahrenheit.

Chapter 17
The Dual-Alternative Decision Structure

17.1 The Dual-Alternative Decision Structure

In contrast to the single-alternative decision structure, this type of decision control structure includes a statement or block of statements on both paths.



If *Boolean_Expression* evaluates to `true`, the statement or block of statements 1 is executed; otherwise, the statement or block of statements 2 is executed.

The general form of the Java statement is

if (*Boolean_Expression*) {

```
    A statement or block of statements 1
```

}
else {

```
    A statement or block of statements 2
```

}

In the next example, the message “You are an adult” is displayed when the user enters a value greater than or equal to 18. The message “You are underage!” is displayed otherwise.

Project_17.1

```
public static void main(String[] args) throws Exception {
   int age;
```

```
   System.out.print("Enter your age: "); age = Integer.parseInt(cin.nextLine());
   if (age >= 18) {
      System.out.println("You are an adult!");
   }
   else {
      System.out.println("You are underage!");
   }
}
```

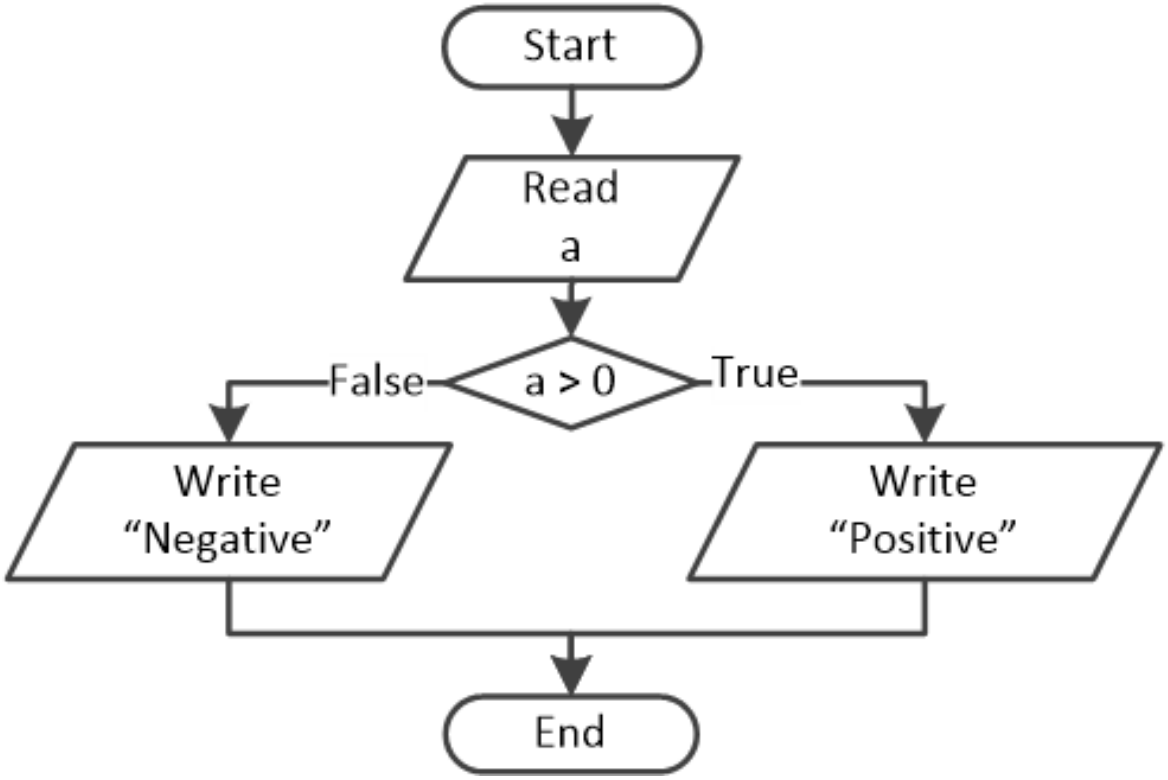
Similar to the single-alternative decision structure, single statements can be written without being enclosed inside braces { }. The `if-else` statement can be written as shown below.

```
if (Boolean_Expression) One_Single_Statement_1; else One_Single_Statement_2;
```

Exercise 17.1-1 Finding the Output Message

For the following flowchart, determine the output message for three different executions.

The input values for the three executions are: (i) 3, (ii) −3, and (iii) 0.



Solution i) When the user enters the value 3, the Boolean expression evaluates to `true`. The flow of execution follows the right path and the message "Positive" is displayed.

ii) When the user enters the value −3, the Boolean expression evaluates to `false`. The flow of execution follows the left path and the message "Negative" is displayed.

iii) Can you predict what happens when the user enters the value 0? If you believe that none of the messages will be displayed, you are wrong! The dual-alternative decision structure must always follow a path, either the right or the left! It cannot skip the execution of both of its blocks of statements. At least one statement or block of statements must be executed. So, in this case, when the user enters the value 0, the Boolean expression evaluates to `false`, the flow of execution follows the left path, and the message "Negative" is displayed!

This algorithm, as it stands, does not satisfy the property of effectiveness for all possible inputs. While it correctly identifies positive and negative values, it overlooks the case of zero. Zero is a valid input and should be accounted for in the algorithm's logic to ensure it produces a meaningful result for all potential inputs. Later in this book (in Exercise 20.1-2), you will learn how to display three messages, depending on whether the user-provided value is greater than, less than, or equal to zero.

A Decision symbol has one entrance and two exit paths! You cannot have a third exit!

Exercise 17.1-2 Trace Tables and Dual-Alternative Decision Structures

Create a trace table to determine the values of the variables in each step of the next Java program for two different executions.

The input values for the two executions are: (i) 5, and (ii) 10.

Project_17.1-2

```
public static void main(String[] args) throws Exception {
   double a, z, w, y;
   a = Double.parseDouble(cin.nextLine());
   z = a * 10; w = (z - 4) * (a - 3) / 7 + 36;
   if (z >= w && a < z) {
      y = 2 * a;
   }
   else {
      y = 4 * a;
   }
   System.out.println(y); }
```

Solution i) For the input value of 5, the trace table looks like this.

Step	Statement	Notes	a	z	w	y
1	`a = Double.parseDouble(…`	User enters the value 5	**5.0**	?	?	?
2	`z = a * 10`		5.0	**50.0**	?	?
3	`w = (z - 4) * (a - 3) / 7 + 36`		5.0	50.0	**49.142**	?
4	`if (z >= w && a < z)`	This evaluates to `true`				
5	`y = 2 * a`		5.0	50.0	49.142	**10.0**
6	`.println(y)`	It displays: 10.0				

ii) For the input value of 10, the trace table looks like this.

Step	Statement	Notes	a	z	w	y
1	`a = Double.parseDouble(…`	User enters the value 10	**10.0**	?	?	?
2	`z = a * 10`		10.0	**100.0**	?	?
3	`w = (z - 4) * (a - 3) / 7 + 36`		10.0	100.0	**132.0**	?
4	`if (z >= w && a < z)`	This evaluates to `false`				
5	`y = 4 * a`		10.0	100.0	132.0	**40.0**
6	`.println(y)`	It displays: 40.0				

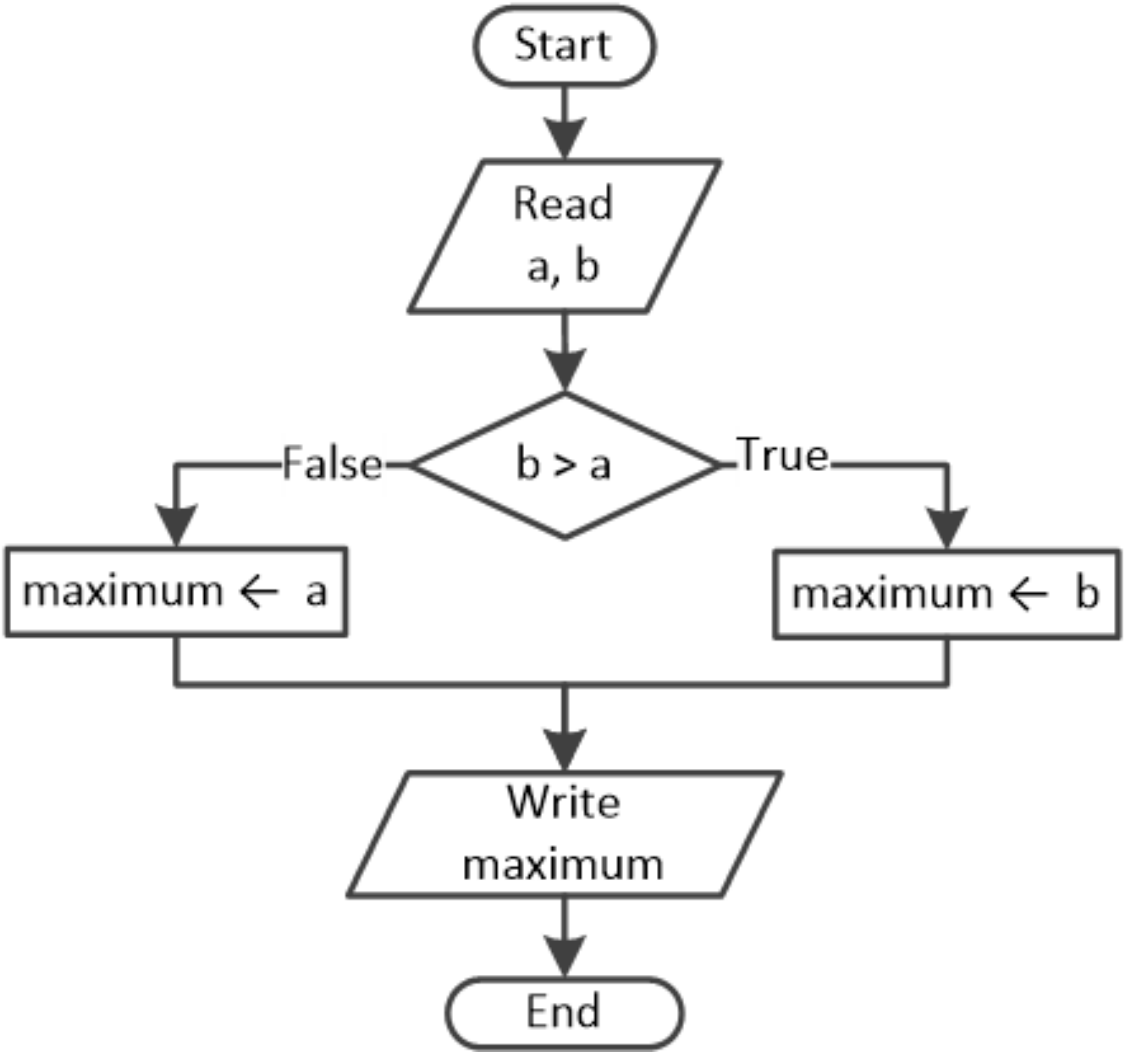
Exercise 17.1-3 Who is the Greatest?

Design a flowchart and write the corresponding Java program that lets the user enter two numbers A and B and then determines and displays the greater of the two numbers.

Solution This exercise can be solved using either the dual- or single-alternative decision structure. So, let's use them both!

First approach – Using a dual-alternative decision structure This approach tests if the value of number B is greater than that of number A.

If so, number B is the greatest; otherwise, number A is the greatest. The corresponding flowchart for solving this exercise using this approach is presented below.



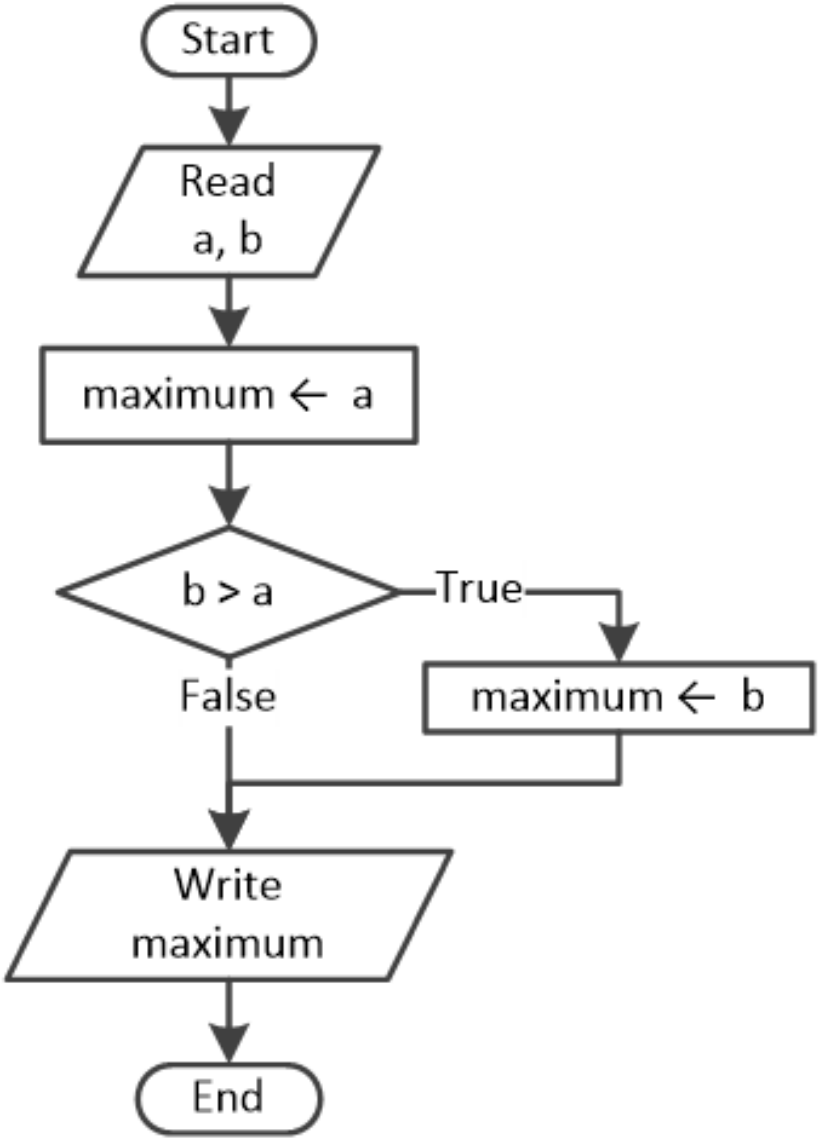
and the Java program is as follows.

```
                    Project_17.1-3a
public static void main(String[] args) throws Exception {
                    double a, b, maximum;
        a = Double.parseDouble(cin.nextLine()); b =
            Double.parseDouble(cin.nextLine());
                        if (b > a) {
                          maximum = b;
                              }
                           else {
```

```
                maximum = a;
                }
    System.out.println("Greatest value: " + maximum); }
```

✎ *Note that this exercise is trying to determine the greatest value and not which variable this value is actually assigned to (to variable A or to variable B).*

Second approach – Using a single-alternative decision structure As presented in the following flowchart, this approach initially assumes that number A is likely the greatest value (this is why it assigns the value of variable `a` to variable `maximum`). However, if it turns out that number B is greater than number A, then the greatest value is updated; variable `maximum` is assigned a new value—the value of variable `b`. Thus, irrespective of the values of numbers A and B, in the end, variable `maximum` will always contain the greatest value!



The Java program is shown here.

```
                    Project_17.1-3b
public static void main(String[] args) throws Exception {
                    double a, b, maximum;
          a = Double.parseDouble(cin.nextLine()); b =
              Double.parseDouble(cin.nextLine());
                    maximum = a; if (b > a) {
                            maximum = b;
                                }
     System.out.println("Greatest value: " + maximum); }
```

Exercise 17.1-4 Finding Odd and Even Numbers

Design a flowchart and write the corresponding Java program that prompts the user to enter a positive integer, and then displays a message indicating whether this number is even; it must display "Odd" otherwise.

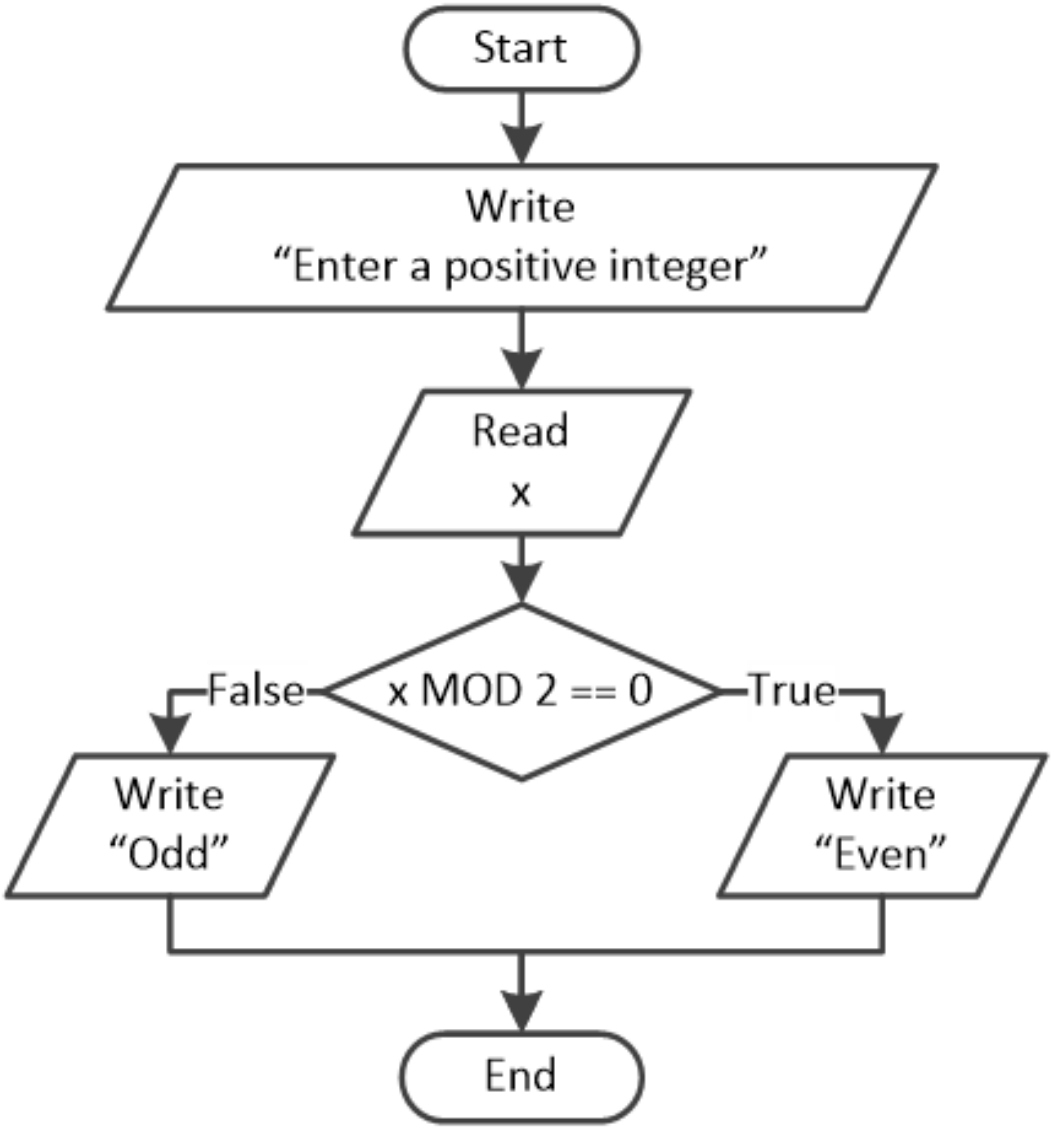
Solution Next you can find various odd and even numbers: ► Odd numbers: 1, 3, 5, 7, 9, 11, …

► Even numbers: 0, 2, 4, 6, 8, 10, 12, ….

✎ *Note that zero is considered an even number.*

In this exercise, you need to find a way to determine whether a number is odd or even. You need to find a common attribute between all even numbers, or between all odd numbers. And actually there is one! All even numbers are exactly divisible by 2. So, when the result of the operation `x MOD 2` equals 0, `x` is even; otherwise, `x` is odd.

The flowchart is shown here.



and the Java program is as follows.

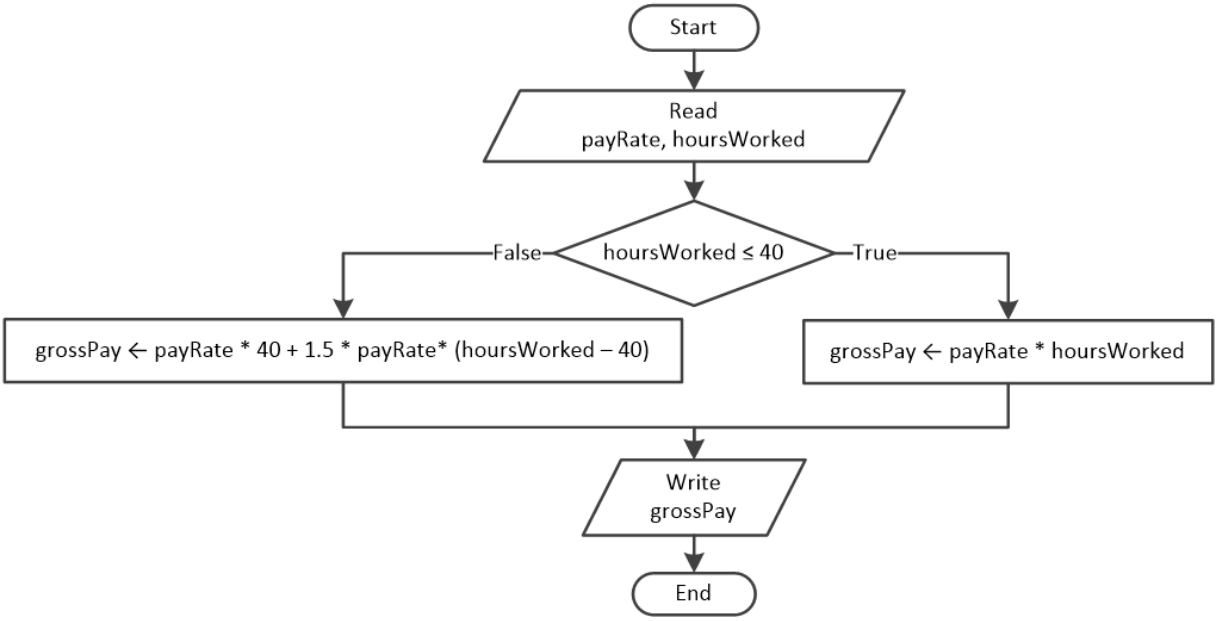
Project_17.1-4

```
public static void main(String[] args) throws Exception {
   int x;
   System.out.print("Enter a positive integer: "); x = Integer.parseInt(cin.nextLine());
   if (x % 2 == 0) {
      System.out.println("Even");
   }
   else {
      System.out.println("Odd");
   }
}
```

Exercise 17.1-5 Weekly Wages

Gross pay depends on the pay rate and the total number of hours worked per week. However, if someone works more than 40 hours, they get paid time-and-a-half for all hours worked over 40. Design a flowchart and write the corresponding Java program that lets the user enter a pay rate and the hours worked and then calculates and displays the gross pay.

Solution This exercise can be solved using the dual-alternative decision structure. When the hours worked are over 40, the gross pay is calculated as follows: gross pay = (pay rate) × 40 + 1.5 × (pay rate) × (all hours worked over 40) The flowchart that solves this problem is shown here.



and the Java program is shown here.

Project_17.1-5

```
public static void main(String[] args) throws Exception {
   int hoursWorked; double payRate, grossPay;
   payRate = Double.parseDouble(cin.nextLine()); hoursWorked =
   Integer.parseInt(cin.nextLine());
   if (hoursWorked <= 40) {
      grossPay = payRate * hoursWorked;
   }
   else {
      grossPay = payRate * 40 + 1.5 * payRate * (hoursWorked - 40);
   }
   System.out.println("Gross Pay: " + grossPay); }
```

17.2 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) It is a possible that none of the statements enclosed in a dual-alternative decision structure will be executed.
2) The dual-alternative decision structure must include at least two statements.
3) The dual-alternative decision structure uses the reserved keyword `else`.
4) The following statement is syntactically correct.

```
int else = 5;
```

5) In a dual-alternative decision structure, the evaluated Boolean expression can return more than two values.
6) The following code fragment satisfies the property of effectiveness.

```
int x, y, z;
x = Integer.parseInt(cin.nextLine()); y = Integer.parseInt(cin.nextLine()); z =
Integer.parseInt(cin.nextLine());
if (x > y && x > z) {
   System.out.println("Value " + x + " is the greatest one"); }
else {
   System.out.println("Value " + y + " is the greatest one"); }
```

17.3 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The dual-alternative decision structure includes a statement or block of statements on
   a) the false path only.
   b) both paths.
   c) the true path only.
2) In the following code fragment,

```
if (x % 2 == 0) {
      x = 0; }
else {
      y++;
}
```

   the statement `y++` is executed when
   a) variable `x` is exactly divisible by 2.
   b) variable `x` contains an even number.
   c) variable `x` contains an odd number.

d) none of the above

3) In the following code fragment,

```
if (x == 3) x = 5; else x = 7; y++;
```

the statement y++ is executed a) when variable x contains a value of 3.

b) when variable x contains a value other than 3.

c) both of the above

17.4 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step of the next Java program for two different executions. Then, design the corresponding flowchart.

   The input values for the two executions are: (i) 3, and (ii) 0.5.

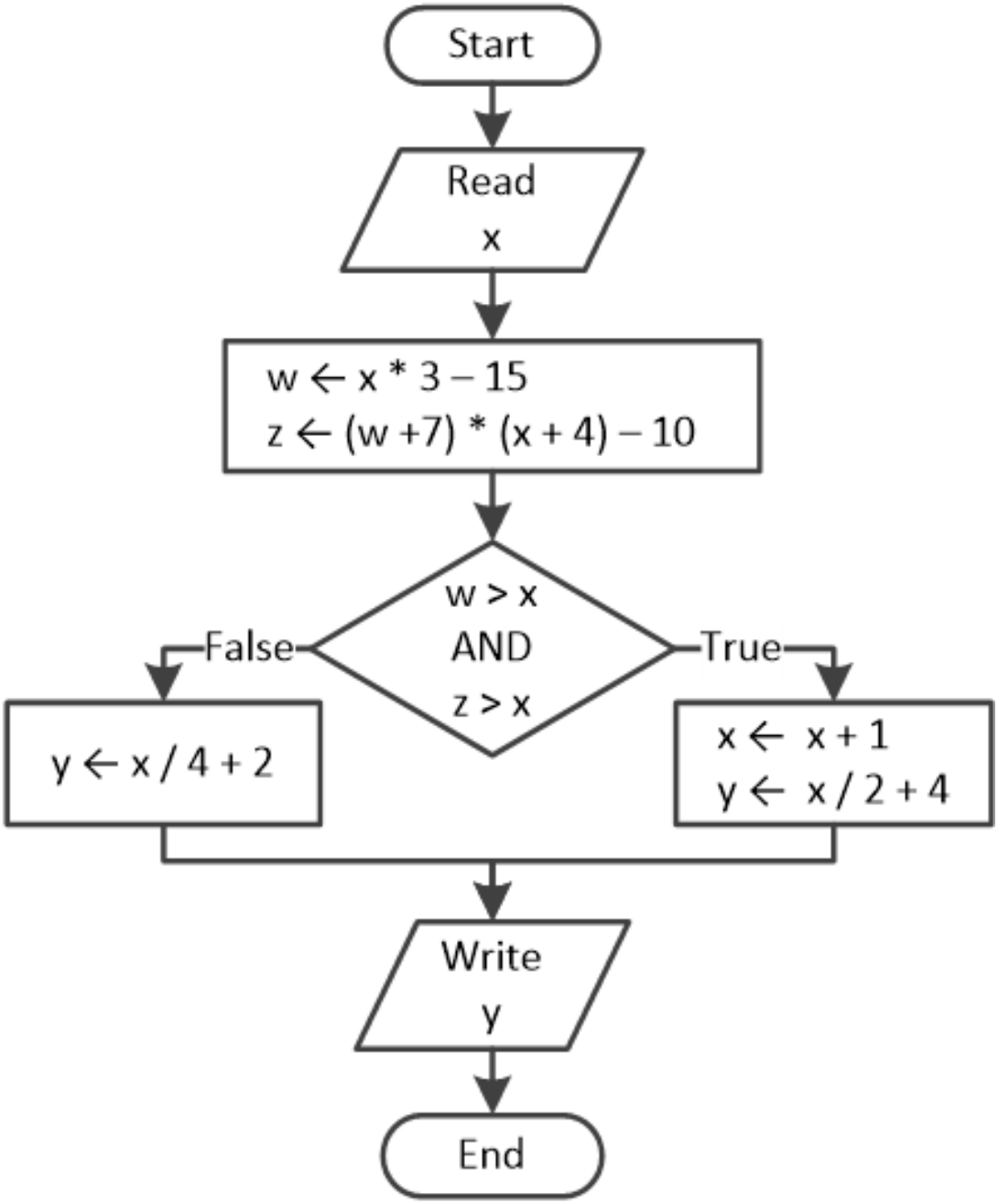
```
public static void main(String[] args) throws Exception {
   double a, z, y;
   a = Double.parseDouble(cin.nextLine()); z = a * 3 - 2; if (z >= 1) {
      y = 6 * a;
   }
   else {
      z++;
      y = 6 * a + z;
   }
   System.out.println(z + ", " + y); }
```

2) Create a trace table to determine the values of the variables in each step of the next Java program. Then, design the corresponding flowchart.

```
public static void main(String[] args) throws Exception {
   double x, y, z;
   x = 3; y = Math.pow(x, 3) + 9; z = 2 * x + y - 4; if (x > y) {
      y = z % x;
      z = Math.sqrt(x);
   }
   else {
      x = z % y;
      z = Math.sqrt(y);
   }
   System.out.println(x + ", " + y + ", " + z); }
```

3) Write the Java program that corresponds to the following flowchart and then create a trace table to determine the values of the variables in each step for two different executions.

   The input values for the two executions are: (i) 10, and (ii) 2.



4) Using a dual-alternative decision structure, write a Java program that prompts the user to enter a number, and then displays a message indicating whether the user-provided number is greater than 100. It must display “Provided number is less than or equal to 100” otherwise.

5) Using a dual-alternative decision structure, write a Java program that prompts the user to enter a number, and then displays a message indicating whether the user-provided number is between 0 and 100. It must display “Provided number is not between 0 and 100” otherwise.

6) Two football teams play against each other in the UEFA Champions League. Using a dual-alternative decision structure, write a Java program that prompts the user to enter the names of the two teams and the goals that each team scored, and then displays the name of the winner. Assume that the user enters valid values and there is no tie (draw).

7) Using a dual-alternative decision structure, write a Java program that lets the user enter an integer, and then displays a message indicating whether the user-provided number is a multiple of 6; it must display "NN is not a multiple of 6" otherwise (where NN is the user-provided number). Assume that the user enters a non-negative[14] value.

8) Using a dual-alternative decision structure, write a Java program that lets the user enter an integer, and then displays one of two possible messages. One message indicates if the user-provided number is a multiple of 6 or a multiple of 7; the other message indicates if the user-provided number is neither a multiple of 6 nor a multiple of 7. Assume that the user enters a non-negative value.

9) Using a dual-alternative decision structure, write a Java program that lets the user enter an integer. The program must then display a message indicating whether the user-provided number is a multiple of 4; it must display "NN is not a multiple of 4" otherwise (where NN is the user-provided number). Additionally, the program must present the structure of the user-provided integer as "NN = QQ x 4 + RR", where QQ is the integer quotient, and RR is the integer remainder when NN is divided by 4. For example, if the user-provided integer is 14, the message "14 = 3 x 4 + 2" must be displayed. Assume that the user enters a non-negative value.

10) Using a dual-alternative decision structure, design a flowchart and write the corresponding Java program that lets the user enter two values, and then determines and displays the smaller of the two values. Assume that the user enters two different values.

11) Using a dual-alternative decision structure, write a Java program that lets the user enter three numbers, and then displays a message indicating whether the user-provided numbers can be lengths of the three sides of a triangle; it must display "Provided numbers cannot be lengths of the three sides of a triangle" otherwise. Assume that the user enters valid values.

Hint: In any triangle, the length of each side is less than the sum of the lengths of the other two sides.

12) Using a dual-alternative decision structure, write a Java program that lets the user enter three numbers, and then displays a message indicating whether the user-provided numbers can be lengths of the three sides of a right triangle (or right-angled triangle); it must display "Provided numbers cannot be lengths of the three sides of a right triangle" otherwise. Assume that the user enters valid values.

    Hint 1: Use the Pythagorean theorem.

    Hint 2: You can use lengths of 3, 4 and 5 (which can be lengths of the three sides of a right triangle) to test your program.

13) Athletes in the long jump at the Olympic Games in Athens in 2004 participated in three different qualifying jumps. An athlete, in order to qualify, has to achieve an average jump distance of at least 8 meters. Using a dual-alternative decision structure, write a Java program that prompts the user to enter the three performances, and then displays the message "Qualified" when the average value is greater than or equal to 8 meters; it displays "Disqualified" otherwise. Assume that the user enters valid values.

14) Gross pay depends on the pay rate and the total number of hours worked per week. However, if someone works more than 40 hours, they get paid double for all hours worked over 40. Using a dual-alternative decision structure, design a flowchart and write the corresponding Java program that lets the user enter the pay rate and hours worked and then calculates and displays net pay. Net pay is the amount of pay that is actually paid to the employee after any deductions. Deductions include taxes, health insurance, retirement plans, on so on. Assume a total deduction of 30%. Also, assume that the user enters valid values.

15) Regular servicing will keep your vehicle more reliable, reducing the chance of breakdowns, inconvenience and unnecessary expenses. In general, there are two types of service you need to perform: a) a minor service every 6000 miles b) a major service every 12000 miles Using a dual-alternative decision structure, write a Java program that prompts the user to enter the miles traveled, and then calculates and displays how many miles are left until the next service, as well as the type of the next service. Assume that the user enters a valid value.

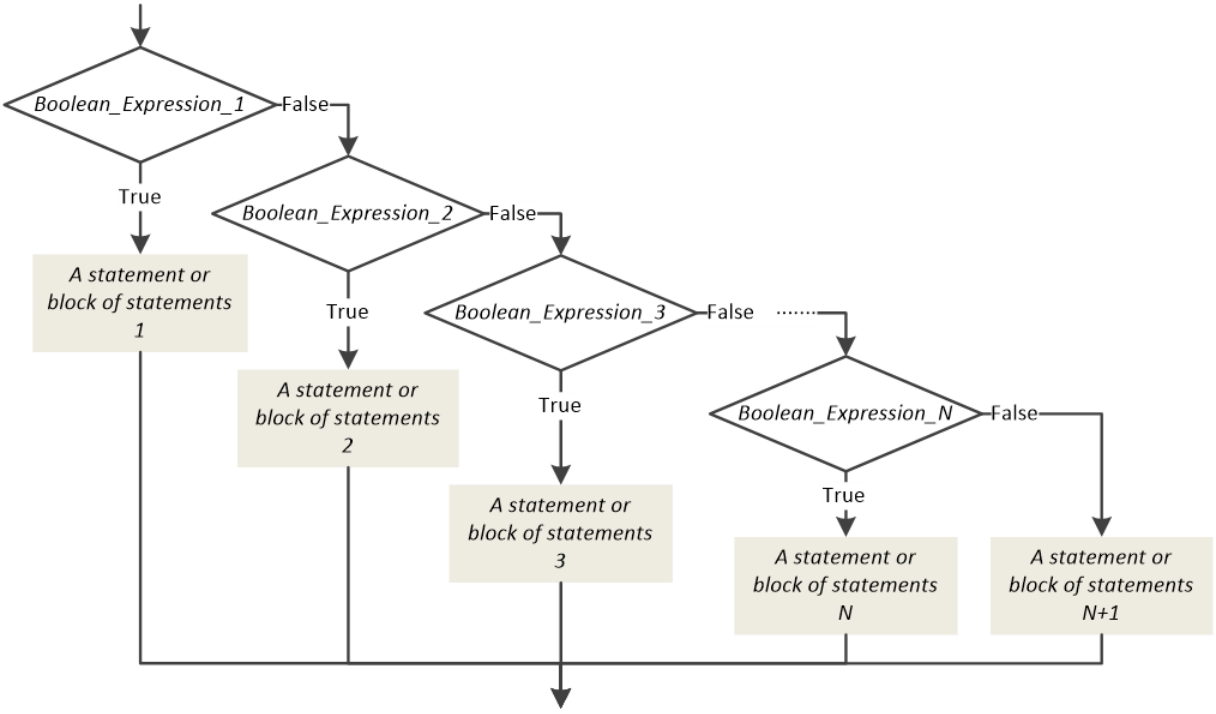
16) Two cars start from rest and move with a constant acceleration along a straight horizontal road for a specified time. Using a dual-alternative decision structure, write a Java program that prompts the user to enter the time the two cars traveled (same for both cars) and the acceleration for each one of them, and then calculates and displays the distance between them as well as a message “Car A is first” or “Car B is first” depending on which car is leading the race. The required formula is $S = u_o + \frac{1}{2}at^2$

where ► ***S*** is the distance the car traveled, in meters (m) ► $\boldsymbol{u_o}$ is the initial velocity (speed) of the car, in meters per second (m/sec) ► ***t*** is the time the car traveled, in seconds (sec) ► ***a*** is the acceleration, in meters per second^2 (m/sec^2) Assume that the user-provided acceleration values are different from each other. Also assume that the user enters valid values.

Chapter 18
The Multiple-Alternative Decision Structure

18.1 The Multiple-Alternative Decision Structure

The multiple-alternative decision structure is used to expand the number of alternatives, as shown in the following flowchart fragment.



When a multiple-alternative decision structure is executed, *Boolean_Expression_1* is evaluated. If it evaluates to `true`, the corresponding statement or block of statements that immediately follows it is executed; then the rest of the structure is skipped, continuing to any remaining statements that may exist **after** the multiple-alternative decision structure. However, if *Boolean_Expression_1* evaluates to `false`, the flow of execution evaluates *Boolean_Expression_2*. If it evaluates to `true`, the corresponding statement or block of statements that immediately follows it is executed and the rest of the structure is skipped. This process continues until one Boolean expression evaluates to `true` or until no more Boolean expressions are left.

The last statement or block of statements N + 1 is executed when none of the previous Boolean expressions has evaluated to `true`. Moreover, this last

statement or block of statements N + 1 is optional and can be omitted. It depends on the algorithm you are trying to solve.

The general form of the Java statement is

if (*Boolean_Expression_1*) {

```
A statement or block of statements 1
```

}
else if (*Boolean_Expression_2*) {

```
A statement or block of statements 2
```

}
else if (*Boolean_Expression_3*) {

```
A statement or block of statements 3
```

}
.
.
.
else if (*Boolean_Expression_N*) {

```
A statement or block of statements N
```

}
else {

```
A statement or block of statements N + 1
```

}

✎ *The last statement or last block of statements N + 1 is optional and can be omitted (you need to omit the keyword* `else` *as well).*

A simple example is shown here.

Project_18.1

```
public static void main(String[] args) throws Exception {
   String name;
   System.out.print("What is your name? "); name = cin.nextLine();
   if (name.equals("John") == true) {
      System.out.println("You are my cousin!");
   }
   else if (name.equals("Aphrodite") == true) {
```

```
      System.out.println("You are my sister!");
   }
   else if (name.equals("Loukia") == true) {
      System.out.println("You are my mom!");
   }
   else {
      System.out.println("Sorry, I don't know you.");
   }
}
```

Exercise 18.1-1 Trace Tables and Multiple-Alternative Decision Structures

Create a trace table to determine the values of the variables in each step for three different executions of the next Java program.

The input values for the three executions are: (i) 5, 8; (ii) −13, 0; and (iii) 1, −1.

Project_18.1-1

```
public static void main(String[] args) throws Exception {
   int a, b;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   if (a > 3)
   System.out.println("Message #1");
   else if (a > 4 && b <= 10) {
      System.out.println("Message #2");
      System.out.println("Message #3");
   }
   else if (a * 2 == -26) {
      System.out.println("Message #4");
      System.out.println("Message #5");
      b++;
   }
   else if (b == 1)
   System.out.println("Message #6");
   else {
      System.out.println("Message #7");
      System.out.println("Message #8");
   }
   System.out.println("The end!"); }
```

✎ *Note that you can use braces { } only when necessary.*

Solution i) For the input values of 5 and 8, the trace table looks like this.

Step	Statement	Notes	a	b

1	`a = Integer.parseI…`	User enters the value 5	**5**	?
2	`b = Integer.parseI…`	User enters the value 8	5	**8**
3	`if (a > 3)`	This evaluates to `true`		
4	`.println("Message #1")`	It displays: Message #1		
5	`.println("The end!")`	It displays: The end!		

✎ *Note that even though the second Boolean expression* `(a > 4 && b <= 10)` *could also have evaluated to* `true`*, it was never checked.*

ii) For the input values of −13 and 0, the trace table looks like this.

Step	Statement	Notes	a	b
1	`a = Integer.parseI…`	User enters the value −13	**−13**	?
2	`b = Integer.parseI…`	User enters the value 0	−13	**0**
3	`if (a > 3)`	This evaluates to `false`		
4	`else if (a > 4 && b <= 10)`	This evaluates to `false`		
5	`else if (a * 2 == -26)`	This evaluates to `true`		
6	`.println("Message #4")`	It displays: Message #4		
7	`.println("Message #5")`	It displays: Message #5		
8	`b++`		−13	**1**
9	`.println("The end!")`	It displays: The end!		

✎ *Note that after step 8 the fourth Boolean expression* `(b == 1)` *could also have evaluated to* `true`*, but it was never checked.*

iii) For the input values of 1 and −1, the trace table looks like this.

Step	Statement	Notes	a	b
1	`a = Integer.parseI…`	User enters the value 1	**1**	?
2	`b = Integer.parseI…`	User enters the value −1	1	**−1**
3	`if (a > 3)`	This evaluates to `false`		

4	`else if (a > 4 && b <= 10)`	This evaluates to `false`
5	`else if (a * 2 == -26)`	This evaluates to `false`
6	`else if (b == 1)`	This evaluates to `false`
7	`.println("Message #7")`	It displays: Message #7
8	`.println("Message #8")`	It displays: Message #8
9	`.println("The end!")`	It displays: The end!

Exercise 18.1-2 Counting the Digits

Write a Java program that prompts the user to enter an integer between 0 and 999 and then counts its total number of digits. In the end, a message "You entered a N-digit number" must be displayed, where N is the total number of digits. Assume that the user enters a valid integer between 0 and 999.

Solution You may be trying to figure out how to solve this exercise using DIV operations. You are probably thinking of dividing the user-provided integer by 10 and checking whether the integer quotient is 0. If it is, this means that the user-provided integer is a one-digit integer. Then, you can divide it by 100 or by 1000 to check for two-digit and three-digit integers, respectively. Your thinking is partly true, and your thoughts are depicted in the following code fragment.

```
if ((int)(x / 10) == 0) digits = 1; else if ((int)(x / 100) == 0) digits = 2; else if
((int)(x / 1000) == 0) digits = 3;
```

If the user-provided integer (in variable `x`) has one digit, the first Boolean expression evaluates to `true` and the rest of the Boolean expressions are never checked! If the user-provided integer has two digits, the first Boolean expression evaluates to `false`, the second one evaluates to `true`, and the last one is never checked! Finally, if the user-provided integer has three digits, both the first and the second Boolean expressions evaluate to `false` and the last one evaluates to `true`!

It seems accurate, doesn't it? So, where does the issue lie?

Consider if the wording of the exercise were "*Write a Java program that prompts the user to enter an integer and displays a message when the user-provided integer consists of two digits*". In all likelihood, you would proceed as follows:

System.out.**print**("*Enter an integer:* "); x = Integer.**parseInt**(cin.**nextLine**());
if ((**int**)(x / 100) == 0) System.out.**println**("*A 2-digit integer entered*");

However, this code is flawed! While the Boolean expression `(int)(x / 100) == 0` works correctly for all user-provided integers with two digits or more, unfortunately, it fails for one-digit integers (as it does not evaluate to `false` for a them). Therefore, using integer division is not the right approach. The correct solution is much simpler than you might believe!

What is the smallest two-digit integer that you can think of? It is 10, right? And what is the greatest one that you can think of? It is 99, right? So, the proper solution is as follows.

```
System.out.print("Enter an integer: "); x = Integer.parseInt(cin.nextLine()); if (x >= 10 && x <= 99) System.out.println("A 2-digit integer entered");
```

According to all these, the complete solution to the exercise is as follows!

```
                    Project_18.1-2a
public static void main(String[] args) throws Exception {
                        int x, digits;
   System.out.print("Enter an integer (0 - 999): "); x =
                Integer.parseInt(cin.nextLine());
                    if (x >= 0 && x <= 9) {
                            digits = 1;
                               }
                else if (x >= 10 && x <= 99) {
                            digits = 2;
                               }
                             else {
                            digits = 3;
                               }
   System.out.println("You entered a " + digits + "-digit
                            number"); }
```

And, if you wish to make your program even better and display an error message to the user when they enter a value that is not between 0 and 999, you can do something like this: Project_18.1-2b

public static void main(**String**[] args) throws Exception {
    int x;
    System.out.**print**("*Enter an integer (0 - 999):* "); x =
    Integer.**parseInt**(cin.**nextLine**());

```
    if (x >= 0 && x <= 9) {
        System.out.println("A 1-digit integer entered");
    }
    else if (x >= 10 && x <= 99) {
        System.out.println("A 2-digit integer entered ");
    }
    else if (x >= 100 && x <= 999) {
        System.out.println("A 3-digit integer entered ");
    }
    else {
        System.out.println("Wrong integer");
    }
}
```

18.2 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) The multiple-alternative decision structure is used to expand the number of alternatives.
2) The multiple-alternative decision structure can have at most three alternatives.
3) In a multiple-alternative decision structure, once a Boolean expression evaluates to `true`, the next Boolean expression is also evaluated.
4) In a multiple-alternative decision structure, the last statement or block of statements N+1 (appearing below the `else` keyword) is always executed.
5) In a multiple-alternative decision structure, the last statement or block of statements N+1 (appearing below the `else` keyword) is executed when at least one of the previous Boolean expressions has evaluated to `true`.
6) In a multiple-alternative decision structure, the last statement or block of statements N+1, and by extension the `else` keyword, can be omitted.
7) In the following code fragment, the statement `y++` is executed only when variable a contains a value other than 1, 2, or 3.

```
if (a == 1) x += 5; else if (a == 2) x -= 2; else if (a == 3) x -= 9; else x += 3;
y++;
```

8) In the code fragment of the previous exercise, the statement `x += 3` is executed only when variable a contains a value other than 1, 2, or 3.

18.3 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step for four different executions of the next Java program.

   The input values for the four executions are: (i) 5, (ii) 150, (iii) 250, and (iv) −1.

```
public static void main(String[] args) throws Exception {
   int q, b;
   q = Integer.parseInt(cin.nextLine());
   if (q > 0 && q <= 50) {
      b = 1;
   }
   else if (q > 50 && q <= 100) {
      b = 2;
   }
   else if (q > 100 && q <= 200) {
      b = 3;
   }
   else {
      b = 4;
   }
   System.out.println(b); }
```

2) Create a trace table to determine the values of the variables in each step for three different executions of the next Java program.

   The input values for the three executions are: (i) 5, (ii) 150, and (iii) −1.

```
public static void main(String[] args) throws Exception {
   double amount, discount, payment;
   amount = Double.parseDouble(cin.nextLine()); discount = 0;
   if (amount < 20) {
      discount = 0;
   }
   else if (amount >= 20 && amount < 60) {
      discount = 5;
   }
   else if (amount >= 60 && amount < 100) {
      discount = 10;
   }
   else if (amount >= 100) {
```

```
    discount = 15;
  }
  payment = amount - amount * discount / 100;
  System.out.println(discount + ", " + payment); }
```

3) Write the following Java program using correct indentation.

```
public static void main(String[] args) throws Exception {
double a, y;
a = Double.parseDouble(cin.nextLine());
if (a < 1) {
y = 5 + a;
System.out.println(y); }
else if (a < 5) {
y = 23 / a;
System.out.println(y); }
else if (a < 10) {
y = 5 * a;
System.out.println(y); }
else {
System.out.println("Error!"); }
}
```

4) Write a Java program that prompts the user to enter two integers and then displays a message indicating whether both numbers are odd or both are even; otherwise the message “Nothing special” must be displayed.

5) Two football teams play against each other in the UEFA Champions League. Write a Java program that prompts the user to enter the names of the two teams and the goals each team scored and then displays the name of the winner or the message “It's a tie!” when both teams score equal number of goals. Assume that the user enters valid values.

6) Design a flowchart and write the corresponding Java program that lets the user enter an integer between −9999 and 9999, and then counts its total number of digits. In the end, a message “You entered a N-digit number” is displayed, where N is the total number of digits. Assume that the user enters a valid integer between −9999 and 9999.

7) Rewrite the Java program of the previous exercise to validate the data input. An error message must be displayed when the user enters an invalid value.

8) Write a Java program that displays the following menu: 1) Convert USD to Euro (EUR) 2) Convert USD to British Pound Sterling (GBP) 3) Convert

USD to Japanese Yen (JPY) 4) Convert USD to Canadian Dollar (CAD) It then prompts the user to enter a choice (of 1, 2, 3, or 4) and an amount in US dollars and calculates and displays the required value. Assume that the user enters valid values. It is given that ► $1 = 0.94 EUR (€) ► $1 = 0.81 GBP (£) ► $1 = ¥ 149.11 JPY

► $1 = 1.36 CAD ($) 9) Write a Java program that prompts the user to enter the number of a month between 1 and 12, and then displays the corresponding season. Assume that the user enters a valid value. It is given that Winter includes months 12, 1, and 2

► Spring includes months 3, 4, and 5

► Summer includes months 6, 7, and 8

► Fall (Autumn) includes months 9, 10, and 11

10) Rewrite the Java program of the previous exercise to validate the data input. An error message must be displayed when the user enters an invalid value.

11) The most popular and commonly used grading system in the United States uses discrete evaluation in the form of letter grades. Design a flowchart and write the corresponding Java program that prompts the user to enter a letter between A and F, and then displays the corresponding percentage according to the following table.

Grade	Percentage
A	90 - 100
B	80 - 89
C	70 - 79
D	60 - 69
E / F	0 - 59

Assume that the user enters a valid value.

12) Write a Java program that prompts the user to enter a number with one decimal digit between 0.0 and 9.9, and then displays the number as English text. For example, if the user enters 2.3, the program must display “Two point three”. Assume that the user enters a valid value.

Hint: Avoid checking each real number individually, as this would require a multiple-alternative decision structure with 100 cases! Try to find a more efficient and clever approach instead!

Chapter 19
The Case Decision Structure

19.1 The Case Decision Structure

The *case decision structure* is a simplified version of the multiple-alternative decision structure. It helps you write code faster and increases readability, especially for algorithms that require complex combinations of decision structures. The case decision structure is used to expand the number of alternatives in the same way as the multiple-alternative decision structure does.

The general form of the Java statement is

```
switch (a variable or an expression to evaluate) {
   case value-1:
      A statement or block of statements 1
      break;
   case value-2:
      A statement or block of statements 2
      break;
   case value-3:
      A statement or block of statements 3
      break;
   .
   .
   .
   case value-N:
      A statement or block of statements N
      break;
   default:
      A statement or block of statements N + 1
}
```

✎ *The last statement or last block of statements N + 1 is optional and can be omitted (you need to omit the keyword* `default` *as well).*

✎ *In order to avoid undesirable results, please remember to always include the keyword* `break` *at the end of each case. If you omit one, two statements or*

blocks of statements are actually executed: the current one in which the keyword `break` *is omitted, and the next one.*

✎ *Note that in Java the* `switch` *statement works only with certain data types such as* `byte`, `short`, `int`, `char`, *or* `String`.

✎ *You cannot always use a case decision structure instead of a multiple-alternative decision structure. In a case decision structure the evaluated variable or expression is written once, which means that this same variable or expression is evaluated in all cases. In a multiple-alternative decision structure, however, the evaluated variable or expression can be different in each case.*

An example that uses the case decision structure is shown here.

Project_19.1

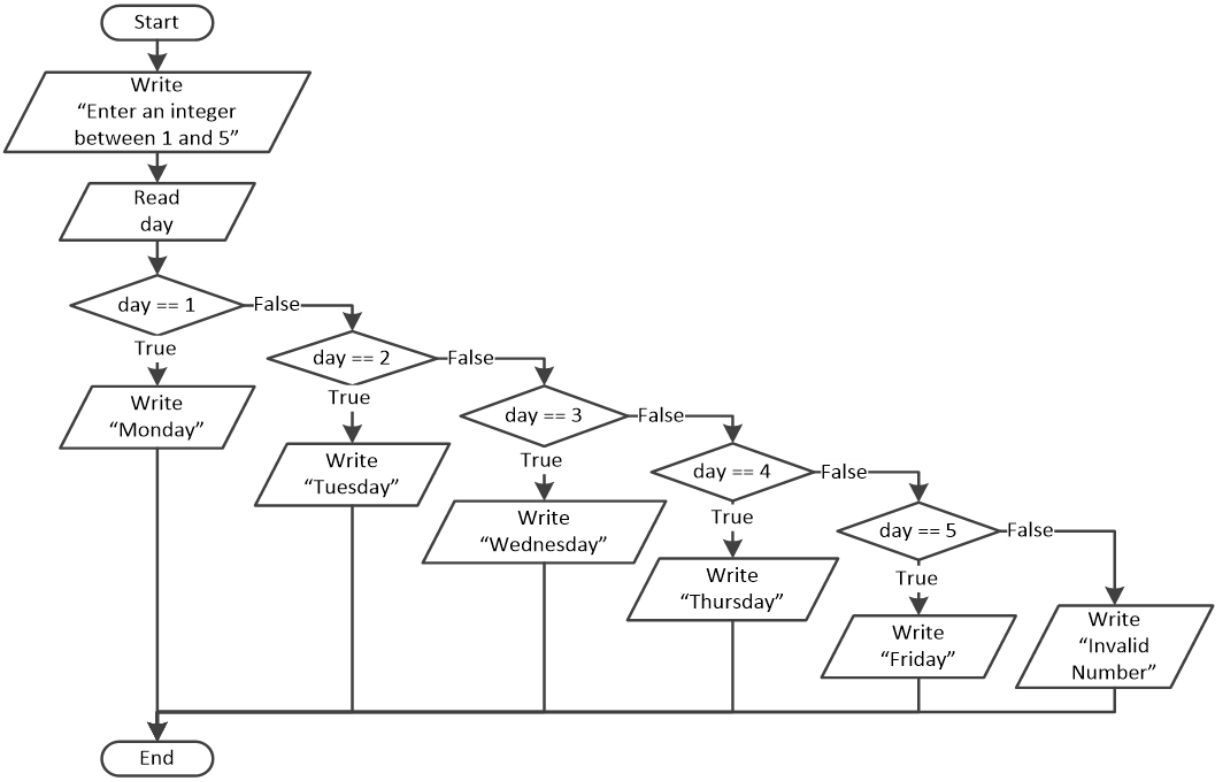
```
public static void main(String[] args) throws Exception {
   String name;
   System.out.print("What is your name? ");
   name = cin.nextLine();
   switch (name) {
      case "John":
         System.out.println("You are my cousin!");
         break;
      case "Aphrodite":
         System.out.println("You are my sister!");
         break;
      case "Loukia":
         System.out.println("You are my mom!");
         break;
      default:
         System.out.println("Sorry, I don't know you.");
   }
}
```

Exercise 19.1-1 The Days of the Week

Write a Java program that prompts the user to enter an integer between 1 and 5, and then displays the corresponding work day (Monday, Tuesday, Wednesday, Thursday, or Friday). If the value entered is invalid, an error message must be displayed.

Solution

The flowchart that solves this exercise is presented below.



The corresponding Java program can be written using either a multiple-alternative decision structure or a case decision structure. Let's try them both!

First approach – Using a multiple-alternative decision structure

Project_19.1-1a

```
public static void main(String[] args) throws Exception {
   int day;
   System.out.print("Enter an integer between 1 and 5: ");
   day = Integer.parseInt(cin.nextLine());
   if (day == 1) {
      System.out.println("Monday");
   }
   else if (day == 2) {
      System.out.println("Tuesday");
   }
   else if (day == 3) {
      System.out.println("Wednesday");
   }
   else if (day == 4) {
      System.out.println("Thursday");
   }
   else if (day == 5) {
```

```
      System.out.println("Friday");
   }
   else {
      System.out.println("Invalid Number");
   }
}
```

Second approach – Using a case decision structure

Project_19.1-1b

```
public static void main(String[] args) throws Exception {
   int day;
   System.out.print("Enter an integer between 1 and 5: ");
   day = Integer.parseInt(cin.nextLine());
   switch (day) {
      case 1:
         System.out.println("Monday");
         break;
      case 2:
         System.out.println("Tuesday");
         break;
      case 3:
         System.out.println("Wednesday");
         break;
      case 4:
         System.out.println("Thursday");
         break;
      case 5:
         System.out.println("Friday");
         break;
      default:
         System.out.println("Invalid Number");
   }
}
```

✎ *The case decision structure and the multiple-alternative decision structure share the same flowchart.*

19.2 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) The case decision structure is used to expand the number of alternatives.
2) The case decision structure can always be used instead of a multiple-alternative decision structure.

3) The case decision structure can have as many alternatives as the programmer wishes.
4) In a case decision structure, the last statement or block of statements N + 1 (appearing below the `default` keyword) is always executed.
5) In a case decision structure, the last statement or block of statements N + 1 (appearing below the `default` keyword) is executed when none of the previous cases has evaluated to `true`.
6) The last statement or block of statements N + 1, as well as the `default` keyword, cannot be omitted.
7) In the following Java program, the statement `y++` is executed only when variable a contains a value other than 1, 2, or 3.

```
switch (a) {
   case 1: x = x + 5; break;
   case 2: x = x - 2; break;
   case 3: x = x - 9; break;
   default: x = x + 3; y++;
}
```

19.3 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step of the next Java program for three different executions.

   The input values for the three executions are: (i) 1, (ii) 3, and (iii) 250.

```
public static void main(String[] args) throws Exception {
   int a, x, y;
   a = Integer.parseInt(cin.nextLine());
   x = 0;
   y = 0;
   switch (a) {
      case 1: x = x + 5; y = y + 5; break;
      case 2: x = x - 2; y--; break;
      case 3: x = x - 9; y = y + 3; break;
      default: x = x + 3; y++;
   }
   System.out.println(x + ", " + y);
}
```

2) Create a trace table to determine the values of the variables in each step of the next Java program for three different executions.

The input values for the three executions are: (i) 10, 2, 5; (ii) 5, 2, 3; and (iii) 4, 6, 2.

```
public static void main(String[] args) throws Exception {
   int a, x;
   double y;
   a = Integer.parseInt(cin.nextLine());
   x = Integer.parseInt(cin.nextLine());
   y = Double.parseDouble(cin.nextLine());
   switch (a) {
      case 10:
         x = x % 2;
         y = Math.pow(y, 2);
         break;
      case 3:
         x = x * 2;
         y--;
         break;
      case 5:
         x = x + 4;
         y += 7;
         break;
      default:
         x -= 3;
         y++;
   }
   System.out.println(x + ", " + y);
}
```

3) Using a case decision structure, write a Java program that prompts the user to enter the name of a month, and then displays the corresponding number (1 for January, 2 for February, and so on). If the value entered is invalid, an error message must be displayed.

4) Using a case decision structure, write a Java program that displays the following menu:

   1) Convert Miles to Yards
   2) Convert Miles to Feet
   3) Convert Miles to Inches

   It then prompts the user to enter a choice (of 1, 2, or 3) and a distance in miles. Then, it calculates and displays the required value. Assume that the user enters a valid value for the distance. However, if the choice entered is invalid, an error message must be displayed. It is given that

- 1 mile = 1760 yards
- 1 mile = 5280 feet
- 1 mile = 63360 inches

5) Roman numerals are shown in the following table.

Number	Roman Numeral
1	I
2	II
3	III
4	IV
5	V
6	VI
7	VII
8	VIII
9	IX
10	X

Using a case decision structure, write a Java program that prompts the user to enter a Roman numeral between I and X, and then displays the corresponding number. However, if the choice entered is invalid, an error message must be displayed.

6) An online supermarket awards points to its customers based on the total number of wine bottles purchased each month. The points are awarded as follows:

- If the customer purchases 1 bottle of wine, they are awarded 3 points.
- If the customer purchases 2 bottles of wine, they are awarded 10 points.
- If the customer purchases 3 bottles of wine, they are awarded 20 points.
- If the customer purchases 4 bottles of wine or more, they are awarded 45 points.

Using a case decision structure, write a Java program that prompts the user to enter the total number of wine bottles they have purchased in a month and then displays the number of points awarded. Assume that the user enters a valid value.

7) Using a case decision structure, write a Java program that prompts the user to enter their name, and then displays “Hello NN!” or “Hi NN!” or “What's up NN!”, where NN is the name of the user. The message to be displayed must be chosen randomly.

8) Using a case decision structure, write a Java program that lets the user enter a word such as "zero", “one” or "two", and then converts it into the corresponding digit, such as 0, 1, or 2. This must be done for the numbers 0 to 9. Display "I don't know this number!" when the user enters an unknown.

9) The Beaufort[15] scale is an empirical measure that relates wind speed to observed conditions on land or at sea. Using a case decision structure, write a Java program that prompts the user to enter the Beaufort number, and then displays the corresponding description from the following table. However, if the number entered is invalid, an error message must be displayed.

Beaufort Number	Description
0	Calm
1	Light air
2	Light breeze
3	Gentle breeze
4	Moderate breeze
5	Fresh breeze
6	Strong breeze
7	Moderate gale
8	Gale
9	Strong gale
10	Storm

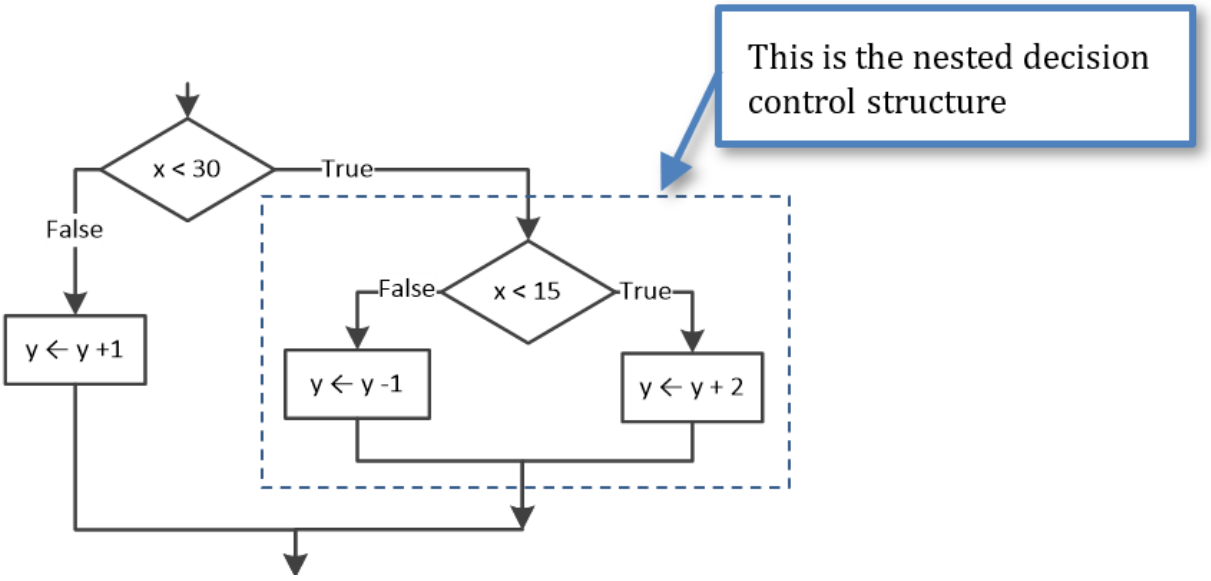
11	Violent storm
12	Hurricane force

Chapter 20
Nested Decision Control Structures

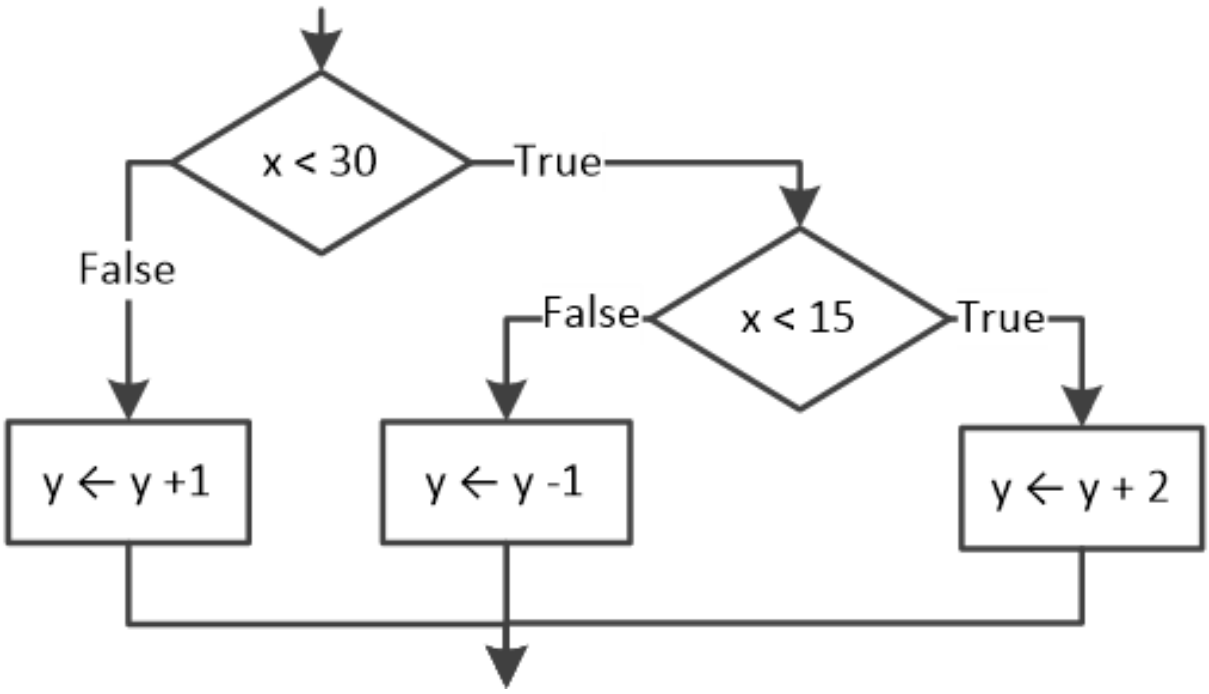
20.1 What are Nested Decision Control Structures?

Nested decision control structures are decision control structures that are "nested" (enclosed) within another decision control structure. This means that one decision control structure can nest (enclose) another decision control structure (which then becomes the "nested" decision control structure). In turn, that nested decision control structure can enclose another decision structure, and so on.

An example of a nested decision control structure is shown here.



This can be rearranged to become



and the Java code is shown here.

```
if (x < 30) {
   if (x < 15) { [More…]
      y = y + 2;
   }
   else {
      y--;
   }
}
else {
   y++;
}
```

There are no practical limitations to how deep this nesting can go. As long as the syntax rules are not violated, you can nest as many decision control structures as you wish. For practical reasons however, as you move to three or four levels of nesting, the entire structure becomes very complex and difficult to understand.

✎ *Complex code may lead to invalid results! Try to keep your code as simple as possible by breaking large nested decision control structures into multiple smaller ones, or by using other types of decision control structures.*

Obviously, you can nest **any** decision control structure inside **any other** decision control structure as long as you keep them syntactically and

logically correct. In the next example, a case decision structure is nested within a dual-alternative decision structure.

Project_20.1

```
public static void main(String[] args) throws Exception {
   int x;
   System.out.print("Enter a choice: ");
   x = Integer.parseInt(cin.nextLine());
   if (x < 1 || x > 4) {
      System.out.println("Invalid choice");
   }
   else {
      System.out.println("Valid choice");
      switch (x) { [More…]
         case 1:
            System.out.println("1st choice selected");
            break;
         case 2:
            System.out.println("2nd choice selected");
            break;
         case 3:
            System.out.println("3rd choice selected");
            break;
         case 4:
            System.out.println("4th choice selected");
            break;
      }
   }
}
```

✎ *Note that keyword default is missing from the switch statement. If you wish to include it, considering that the only choices checked are 1, 2, 3, or 4, you can replace* `case 4` *with* `default`.

Exercise 20.1-1 Trace Tables and Nested Decision Control Structures

Create a trace table to determine the values of the variables in each step of the next Java program for three different executions.

The input values for the three executions are: (i) 13, (ii) 18, and (iii) 30.

Project_20.1-1

```
public static void main(String[] args) throws Exception {
   int x, y;
   x = Integer.parseInt(cin.nextLine());
```

```
    y = 10;
    if (x < 30) {
       if (x < 15) {
          y = y + 2;
       }
       else {
          y--;
       }
    }
    else {
       y++;
    }
    System.out.println(y);
}
```

Solution

i) For the input value of 13, the trace table looks like this.

Step	Statement	Notes	x	y
1	`x = Integer.parseI…`	User enters the value 13	**13**	?
2	`y = 10`		13	**10**
3	`if (x < 30)`	This evaluates to `true`		
4	`if (x < 15)`	This evaluates to `true`		
5	`y = y + 2`		13	**12**
6	`.println(y)`	It displays: 12		

ii) For the input value of 18, the trace table looks like this.

Step	Statement	Notes	x	y
1	`x = Integer.parseI…`	User enters the value 18	**18**	?
2	`y = 10`		18	**10**
3	`if (x < 30)`	This evaluates to `true`		
4	`if (x < 15)`	This evaluates to `false`		
5	`y--`		18	**9**
6	`.println(y)`	It displays: 9		

iii) For the input value of 30, the trace table looks like this.

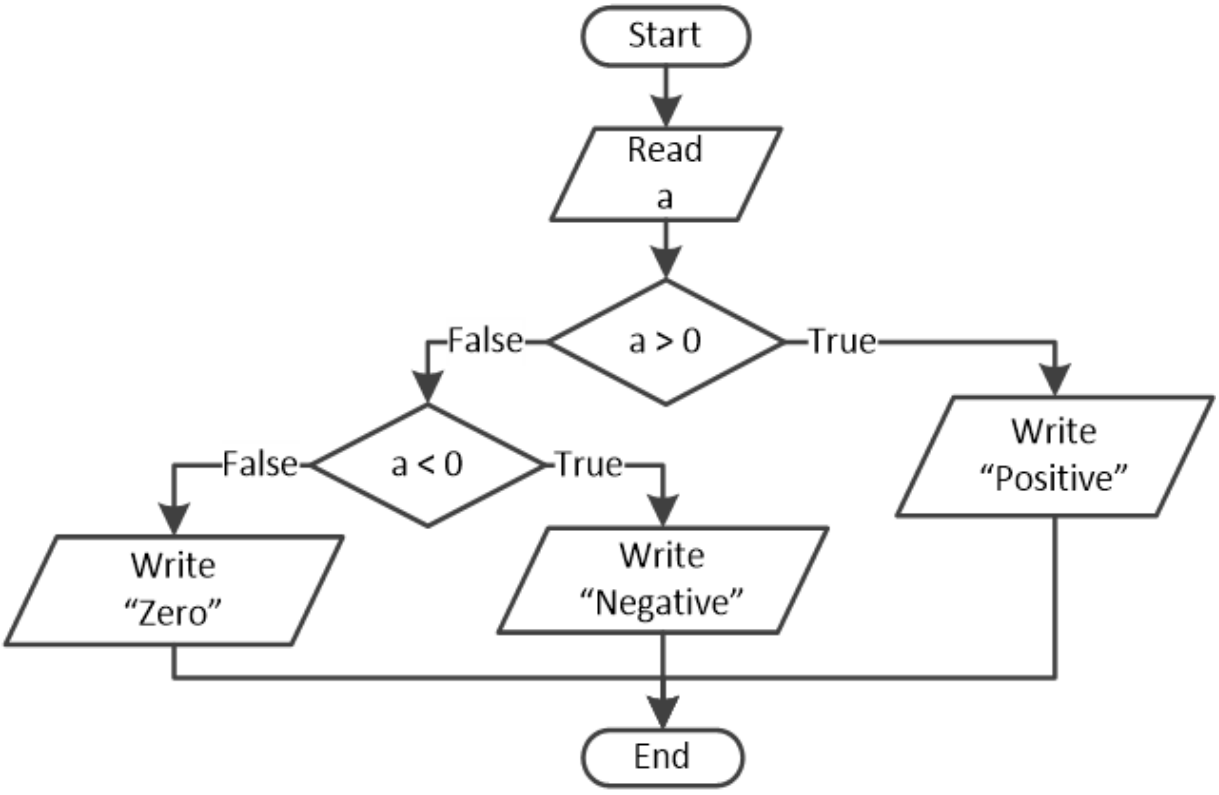
Step	Statement	Notes	x	y
1	`x = Integer.parseI…`	User enters the value 30	**30**	?
2	`y = 10`		30	**10**
3	`if (x < 30)`	This evaluates to `false`		
4	`y++`		30	**11**
5	`.println(y)`	It displays: 11		

Exercise 20.1-2 Positive, Negative or Zero?

Design a flowchart and write the corresponding Java program that lets the user enter a number and then displays the messages "Positive", "Negative", or "Zero" depending on whether the user-provided value is greater than, less than, or equal to zero.

Solution

The flowchart is shown here.



This flowchart can be written as a Java program using either a nested decision control structure or a multiple-alternative decision structure. Let's try them both!

First approach – Using a nested decision control structure

Project_20.1-2a

```
public static void main(String[] args) throws Exception {
   double a;
   a = Double.parseDouble(cin.nextLine());
   if (a > 0) {
      System.out.println("Positive");
   }
   else {
      if (a < 0) {
         System.out.println("Negative");
      }
      else {
         System.out.println("Zero");
      }
   }
}
```

Second approach – Using a multiple-alternative decision structure

Project_20.1-2b

```
public static void main(String[] args) throws Exception {
   double a;
   a = Double.parseDouble(cin.nextLine());
   if (a > 0) {
      System.out.println("Positive");
   }
   else if (a < 0) {
      System.out.println("Negative");
   }
   else {
      System.out.println("Zero");
   }
}
```

20.2 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Nesting of decision control structures describes a situation in which one or more than one path of a decision control structure enclose other decision control structures.

2) Nesting level can go as deep as the programmer wishes.
3) When a problem can be solved using either a case decision structure or nested decision control structures, the second option is better because the program becomes more readable.
4) It is possible to nest a multiple-alternative decision structure within a case decision structure, but not the opposite.

20.3 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step of the next Java program for four different executions.

   The input values for the four executions are: (i) 20, 1; (ii) 20, 3; (iii) 12, 8; and (iv) 50, 0.

```
public static void main(String[] args) throws Exception {
   int x, y;
   x = Integer.parseInt(cin.nextLine());
   y = Integer.parseInt(cin.nextLine());
   if (x < 30) {
      switch (y) {
         case 1:
            x = x % 3;
            y = 5;
            break;
         case 2:
            x = x * 2;
            y = 2;
            break;
         case 3:
            x = x + 5;
            y += 3;
            break;
         default:
            x -= 2;
            y++;
      }
   }
   else {
      y++;
   }
   System.out.println(x + ", " + y);
}
```

2) Create a trace table to determine the values of the variables in each step of the next Java program for four different executions.

The input values for the four executions are: (i) 60, 25; (ii) 50, 8; (iii) 20, 15; and (iv) 10, 30.

```
public static void main(String[] args) throws Exception {
   int x, y;
   x = Integer.parseInt(cin.nextLine());
   y = Integer.parseInt(cin.nextLine());
   if ((x + y) / 2 <= 20) {
      if (y < 10) {
         x = x % 3;
         y += 2;
      }
      else if (y < 20) {
         x = x * 5;
         y += 2;
      }
      else {
         x = x - 2;
         y += 3;
      }
   }
   else {
      if (y < 15) {
         x = x % 4;
         y = 2;
      }
      else if (y < 23) {
         x = x % 2;
         y -= 2;
      }
      else {
         x = 2 * x + 5;
         y += 1;
      }
   }
   System.out.println(x + ", " + y);
}
```

3) Write the following Java program using correct indentation.

```
public static void main(String[] args) throws Exception {
int a;
a = Integer.parseInt(cin.nextLine());
if (a > 1000)
System.out.println("Big Positive");
```

```
else {
if (a > 0)
System.out.println("Positive");
else {
if (a < -1000)
System.out.println("Big Negative");
else {
if (a < 0)
System.out.println("Negative");
else
System.out.println("Zero");
}
}
}
}
```

4) In Greece, you can drive a small scooter when you are at least 16 years old, whereas you can drive a car when you are at least 18 years old. Write a Java program that prompts the user to enter their age and then displays (depending on the user's age) one of the following messages:
   - ► “You cannot drive either a small scooter or a car”, when the user is younger than 16 years old
   - ► “You can drive a small scooter”, when the user is between 16 and 18 years old
   - ► “You can drive a car and a small scooter”, when the user is 18 years old or older

   An error message must be displayed when the user enters an invalid value.

5) A hoverboard factory manager needs a program to calculate the profit or loss the factory makes during the period of one month. Here's some information:
   - ► It costs the factory $150 to build each hoverboard.
   - ► Hoverboards are sold for $250 each.
   - ► The factory pays $1000 for insurance each month for each employee.

   Write a Java program that prompts the user to input the number of hoverboards sold and the number of employees in the company. Depending on the financial performance of the company, the program must then display one of the following messages:

- Profit
- Loss
- Broke even

An error message must be displayed when the user enters a negative number of hoverboards sold or a non-positive[16] number of employees.

6) Write a Java program that prompts the user to enter their name. The program must then select a random integer between 1 and 24 to represent an hour, and then, it must display the message “The hour is HH:00” and, depending on that number, display either “Good morning NN!”, “Good Evening NN!”, “Good Afternoon NN!”, or “Good Night NN!”, where HH is the randomly chosen hour and NN is the name of the user. Solve this exercise twice, once using nested-decision structures and once using a multiple-alternative decision structure.

7) Write a Java program that prompts the user to enter the lengths of three sides of a triangle, and then determines whether or not the user-provided numbers can be lengths of the three sides of a triangle. If the lengths are not valid, a corresponding message must be displayed; otherwise the program must further determine whether the triangle is

   a) equilateral

      Hint: In an equilateral triangle, all sides are equal.

   b) right (or right-angled)

      Hint: Use the Pythagorean Theorem.

   c) not special

   Hint: In any triangle, the length of each side is less than the sum of the lengths of the other two sides.

8) Inside an automated teller machine (ATM) there are notes of $10, $5, and $1. Write a Java program to emulate the way this ATM works. At the beginning, the machine prompts the user to enter the four-digit PIN and then checks for PIN validity (assume “1234” as the valid PIN). If user-provided PIN is correct, the program must prompt the user to enter the amount of money (an integer value) that they want to withdraw and finally it displays the least number of notes the ATM must dispense. For example, if the user enters an amount of $36, the program must display “3 note(s) of $10, 1 note(s) of $5, and 1 note(s) of $1”. Moreover, if the user enters a wrong PIN, the machine will allow them two retries. If the

user enters an incorrect PIN all three times, the message “PIN locked” must be displayed and the program must end. Assume that the user enters a valid value for the amount.

9) Write a Java program that prompts the user to enter two values, one for temperature and one for wind speed. If the temperature is above 75 degrees Fahrenheit, the day is considered hot, otherwise it is cold. If the wind speed is above 12 miles per hour, the day is considered windy, otherwise it is not windy. The program must display one single message, depending on the user-provided values. For example, if the user enters 60 for temperature and 10 for wind speed, the program must display “The day is cold and not windy”. Assume that the user enters valid values.

Chapter 21 More about Flowcharts with Decision Control Structures

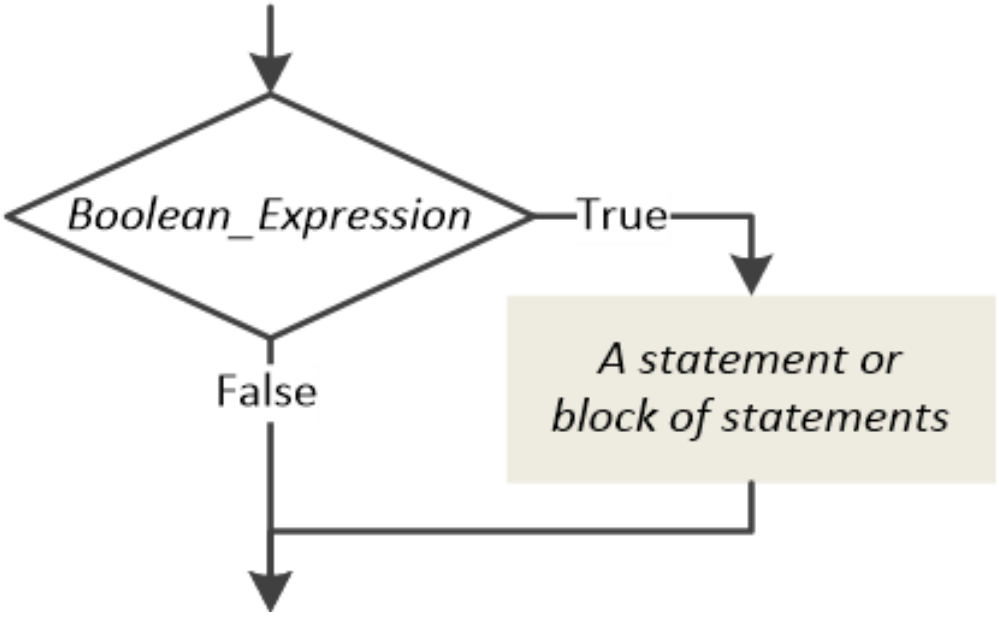
21.1 Introduction

By working through the previous chapters, you've become familiar with all the decision control structures. Since flowcharts are an ideal way to learn "Algorithmic Thinking" and to help you better understand specific control structures, this chapter is dedicated to teaching you how to convert a Java program to a flowchart, or a flowchart to a Java program.

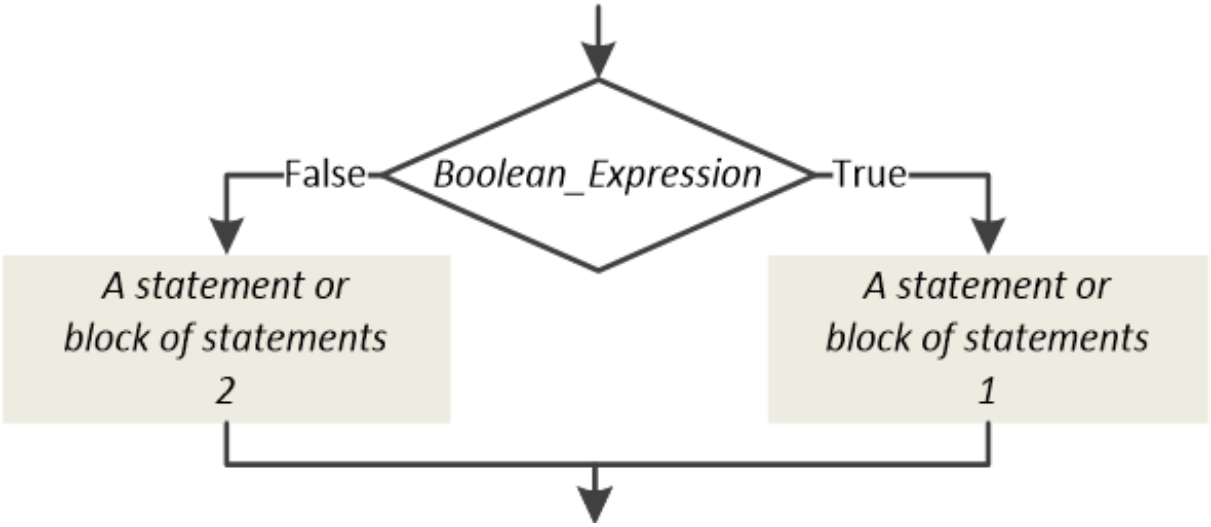
21.2 Converting Java Programs to Flowcharts

To convert a Java program to its corresponding flowchart, you need to recall all the decision control structures and their corresponding flowchart fragments. They are all summarized here.

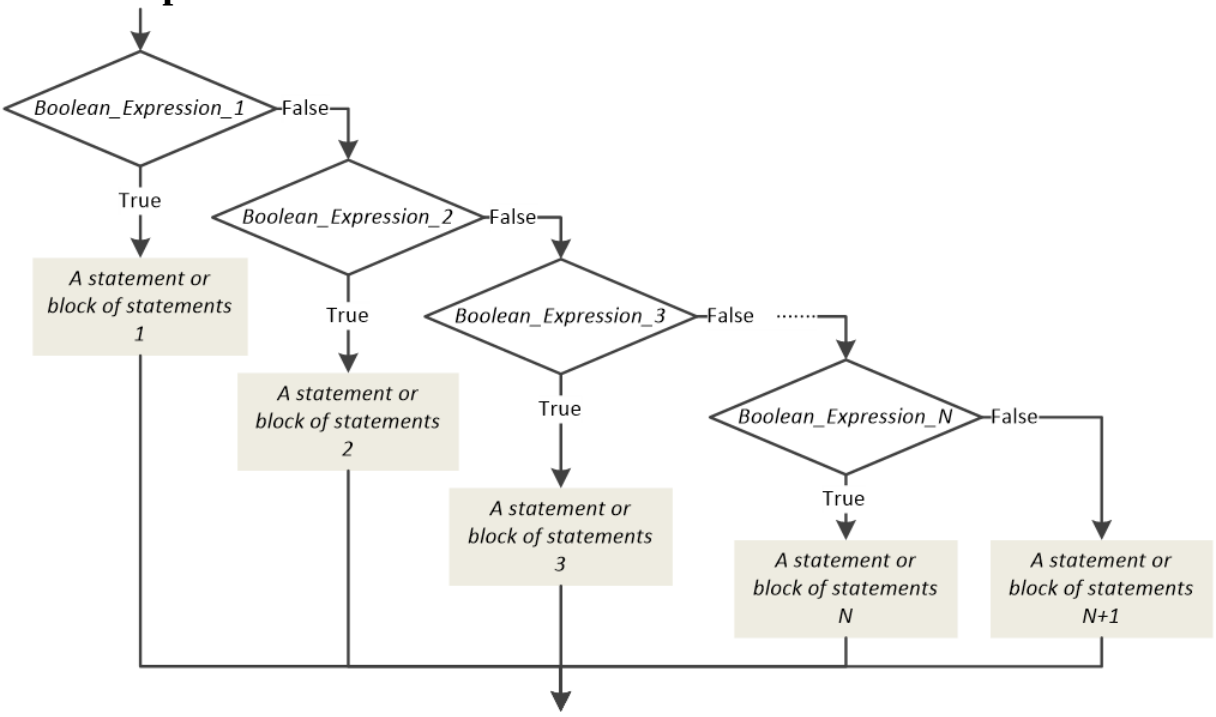
The single-alternative decision structure



The dual-alternative decision structure



The multiple-alternative decision structure



✎ *You can use this same flowchart to represent Java code that uses a case decision structure as well!*

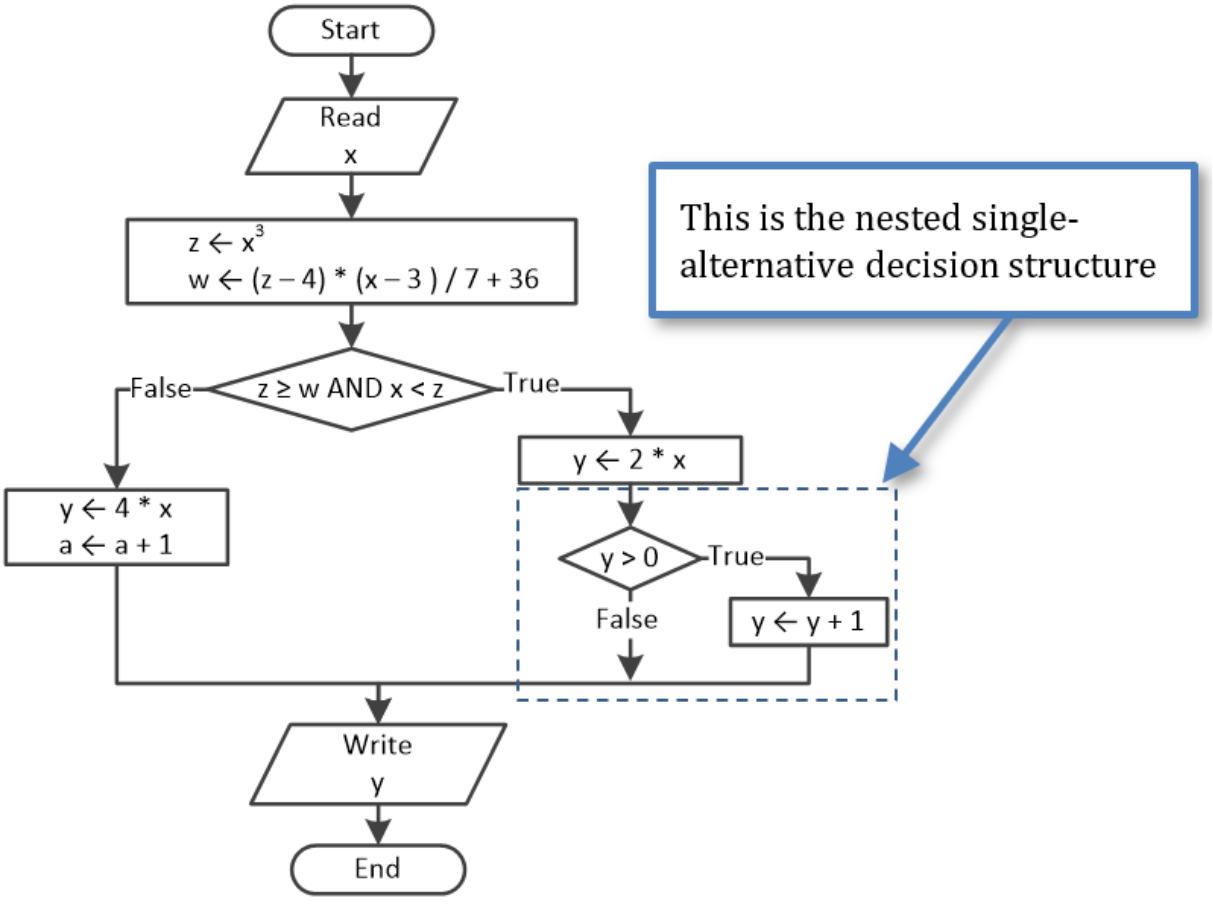
Exercise 21.2-1 Designing the Flowchart

Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
    double x, z, w, y, a;
    x = Double.parseDouble(cin.nextLine());
```

```
z = Math.pow(x, 3); w = (z - 4) * (x - 3) / 7 + 36; if (z >= w && x < z) {
   y = 2 * x;
   if (y > 0) { //This is a nested single-alternative decision structure
      y += 1;
   }
}
else {
   y = 4 * x;
   a++;
}
System.out.println(y); }
```

Solution In this Java program there is a single-alternative decision structure nested within a dual-alternative decision structure. Its corresponding flowchart is as follows.



A flowchart is a very loose method of representing an algorithm. Thus, it is quite permissible to write x^3 or even to use the Java method `Math.pow()`*. Do whatever you wish; everything is permitted, on condition that anyone familiar with flowcharts can clearly understand what you are trying to say!*

In flowcharts, this book uses the commonly accepted AND, OR, and NOT operators!

Exercise 21.2-2 Designing the Flowchart

Design the flowchart that corresponds to the following code fragment given in general form.

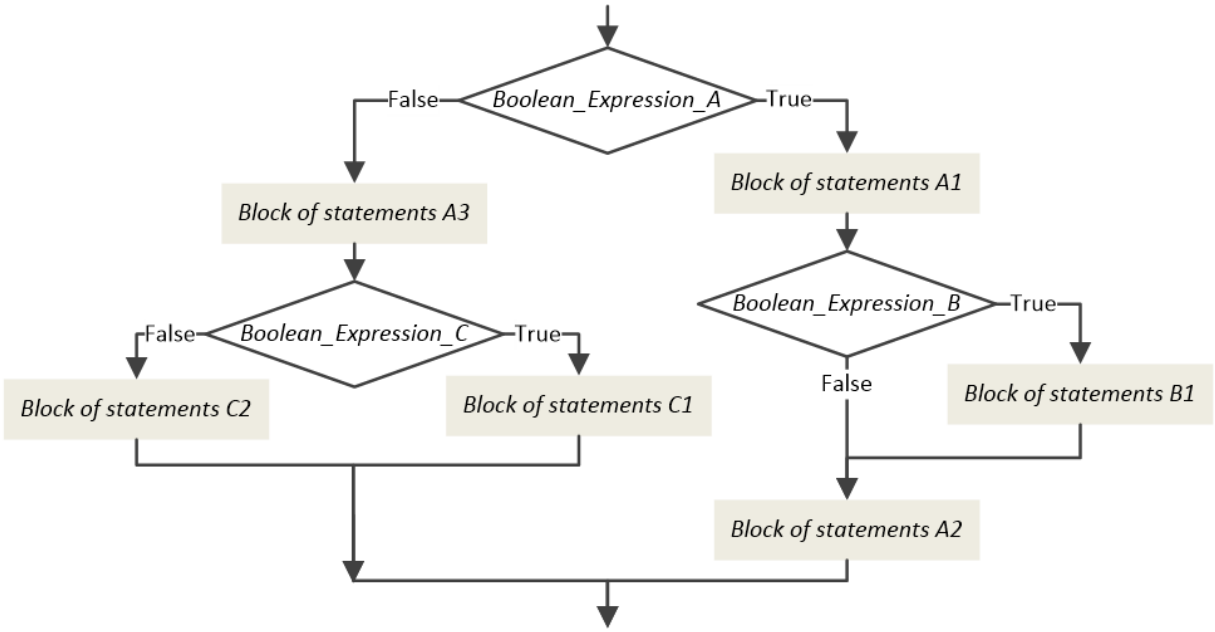
```
if (Boolean_Expression_A) {
    A statement or block of statements A1
    if (Boolean_Expression_B) {
        A statement or block of statements B1
    }
    A statement or block of statements A2
}
else {
    A statement or block of statements A3
    if (Boolean_Expression_C) {
        A statement or block of statements C1
    }
    else {
        A statement or block of statements C2
    }
}
```

Solution For better observation, the initial code fragment is presented again with all the nested decision control structures enclosed in rectangles.

```
if (Boolean_Expression_A) {
    A statement or block of statements A1
    if (Boolean_Expression_B) { [More…]
        A statement or block of statements B1
    }
    A statement or block of statements A2
}
else {
```

```
    A statement or block of statements A3
    if (Boolean_Expression_C) { [More…]
        A statement or block of statements C1
    }
    else {
        A statement or block of statements C2
    }
}
```

and the flowchart fragment in general form is as follows.



Exercise 21.2-3 Designing the Flowchart

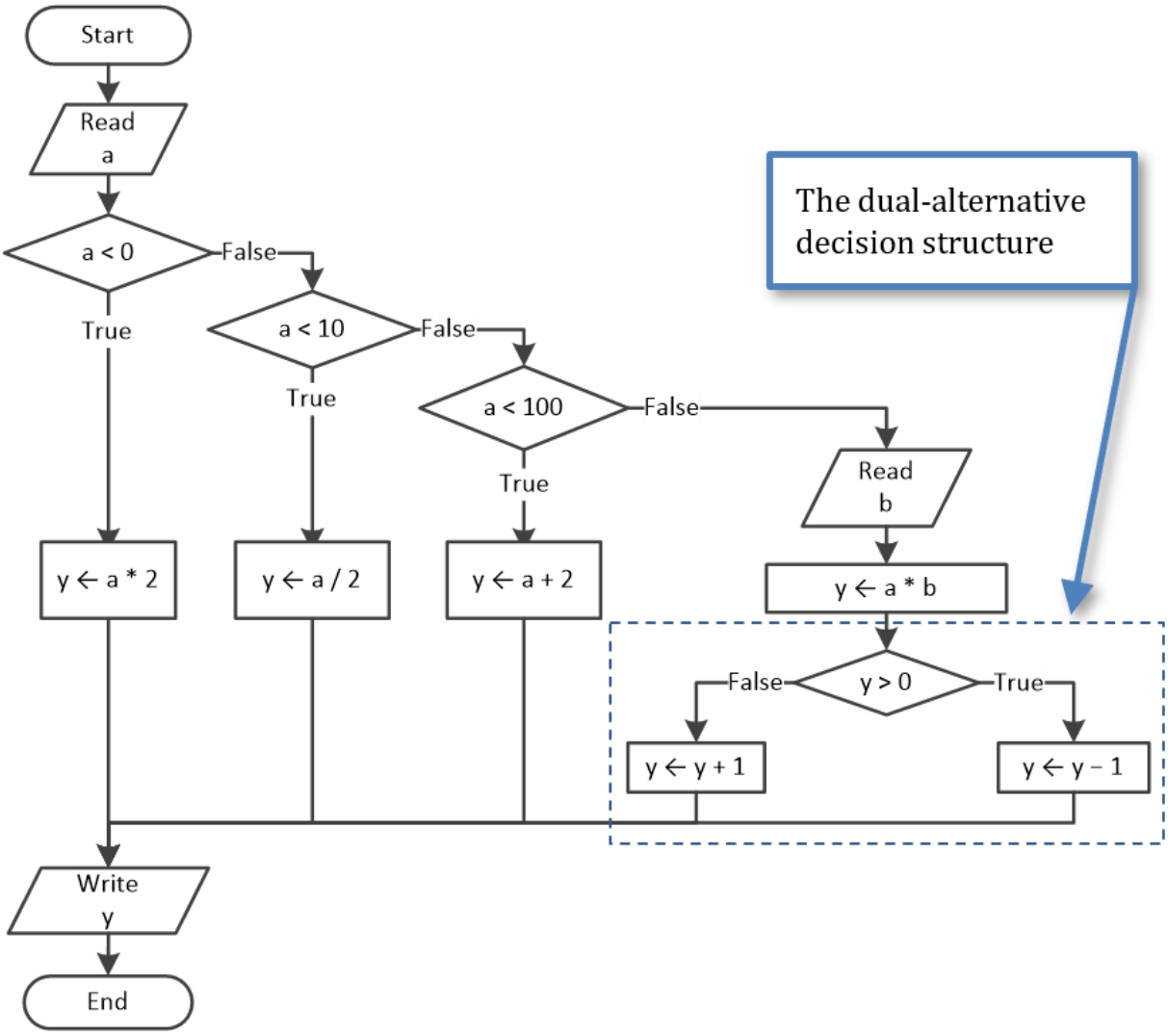
Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
    double a, y, b;
    a = Double.parseDouble(cin.nextLine());
    if (a < 0)
    y = a * 2;
    else if (a < 10)
    y = a / 2;
    else if (a < 100)
    y = a + 2;
    else {
        b = Integer.parseInt(cin.nextLine());
        y = a * b;
```

```
    if (y > 0) //This is a nested dual-alternative decision structure
       y--; //
    else //
       y++; //
}
System.out.println(y); }
```

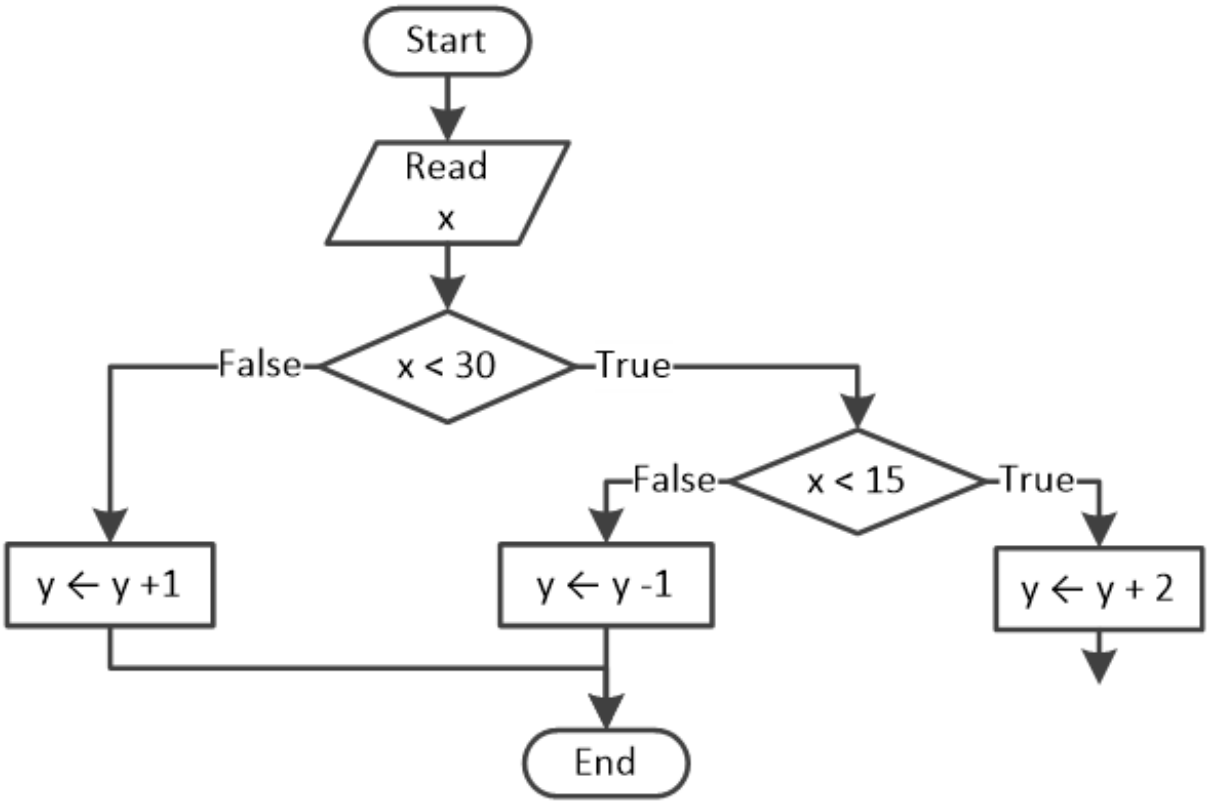
Solution In this Java program, a dual-alternative decision structure is nested within a multiple-alternative decision structure.

The flowchart is shown here.



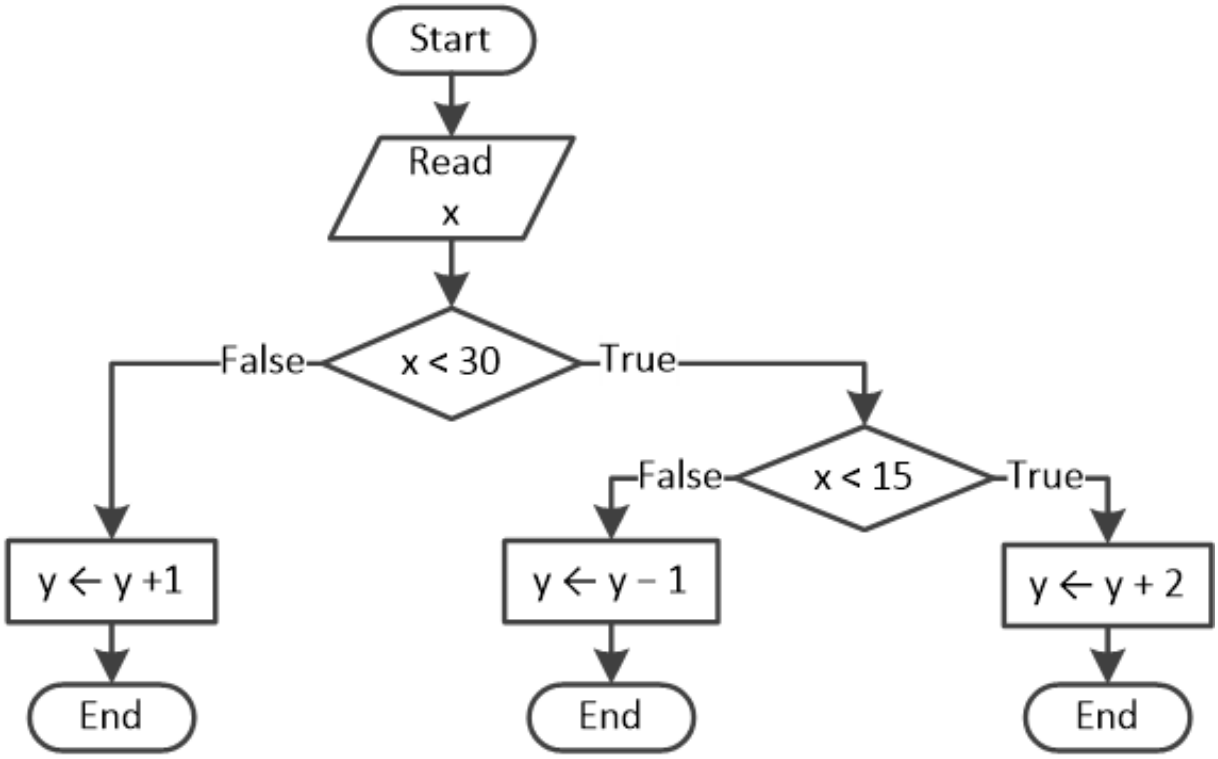
21.3 A Mistake That You Will Probably Make!

In flowcharts, a very common mistake that novice programmers make is to leave some paths unconnected, as shown in the flowchart that follows.

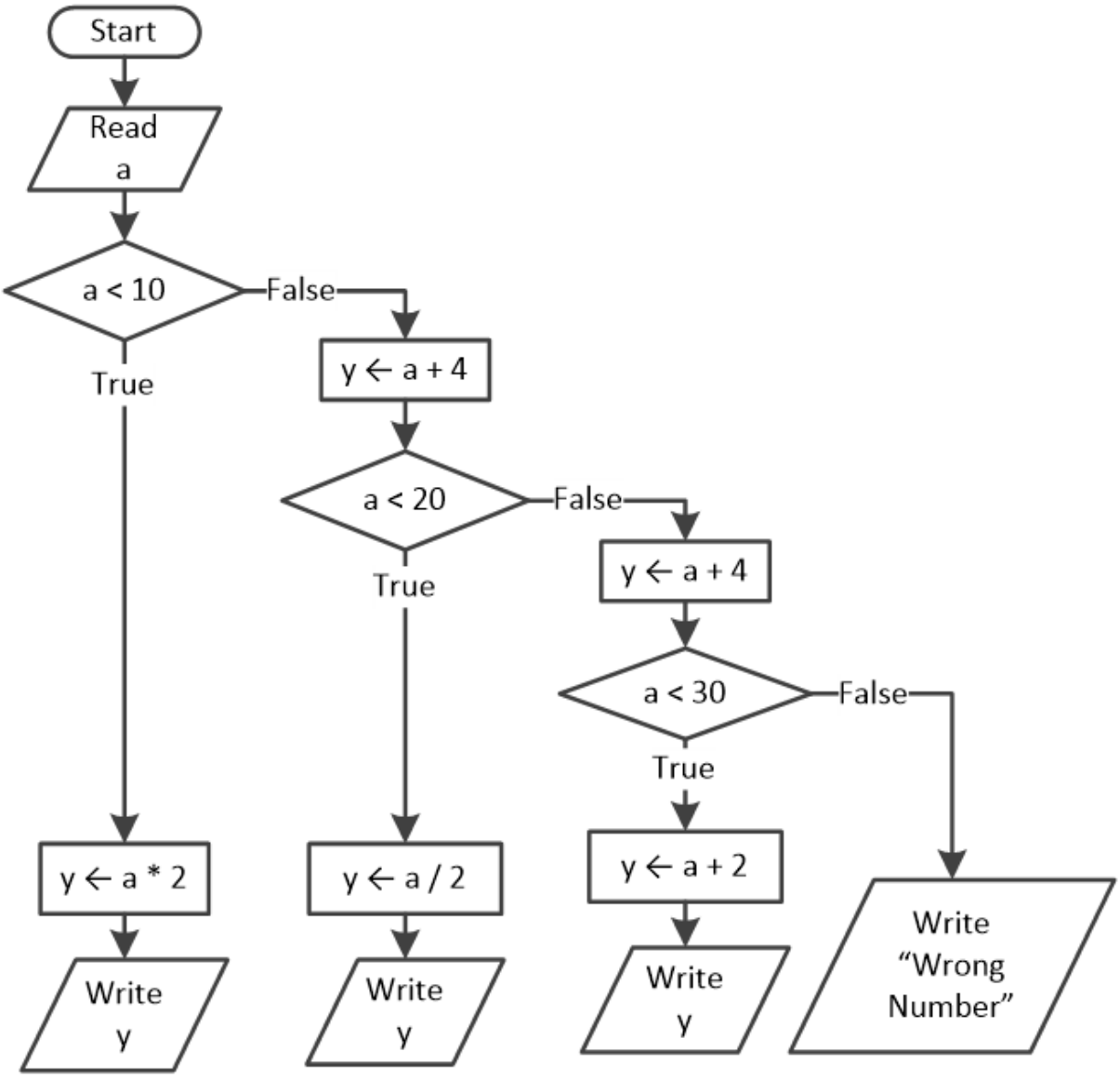


Please keep in mind that every path tries to reach the end of the algorithm, thus you cannot leave any of them unconnected.

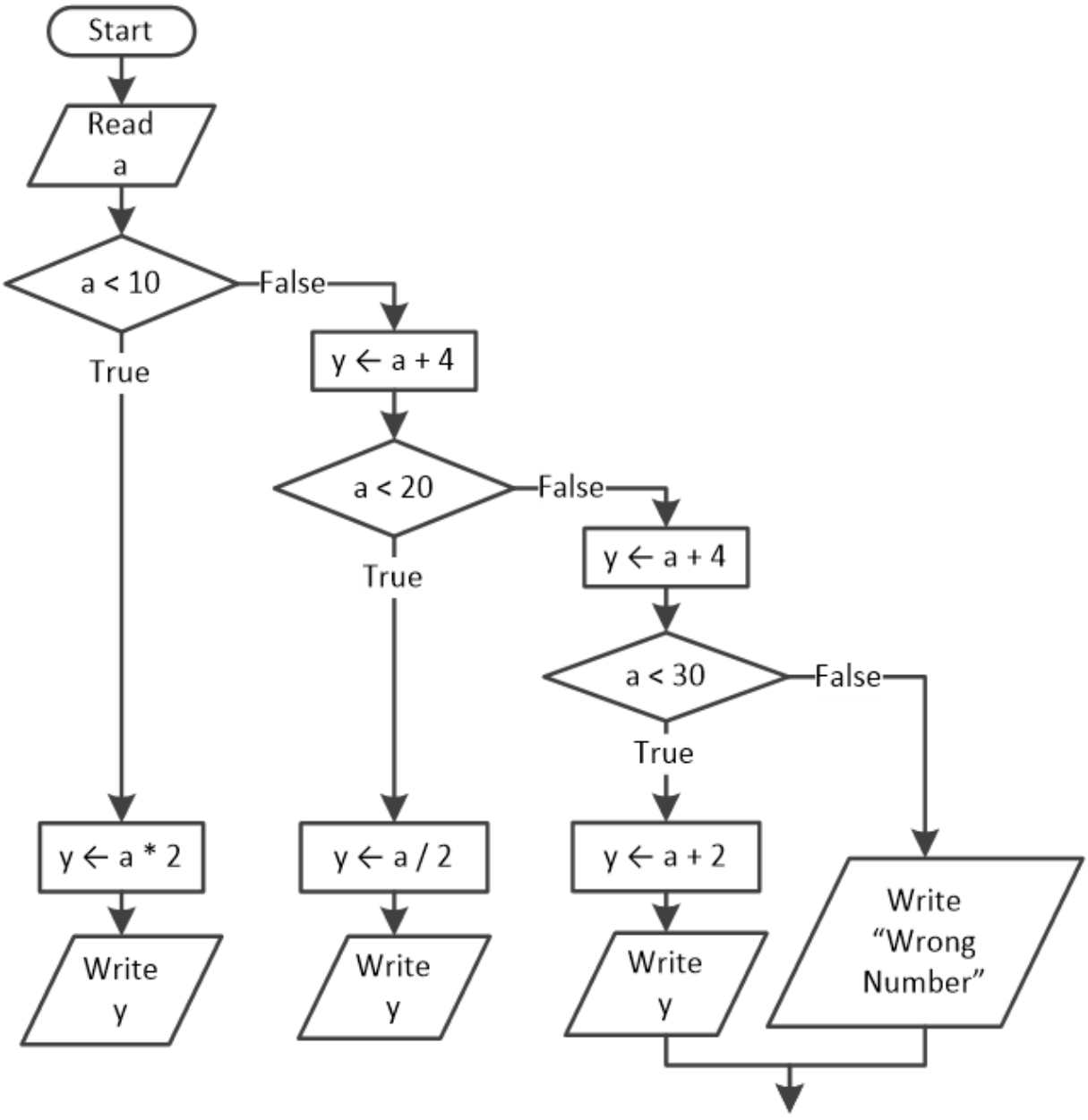
On the other hand, try to avoid flowcharts that use many End symbols, as shown below, since these algorithms are difficult to read and understand.



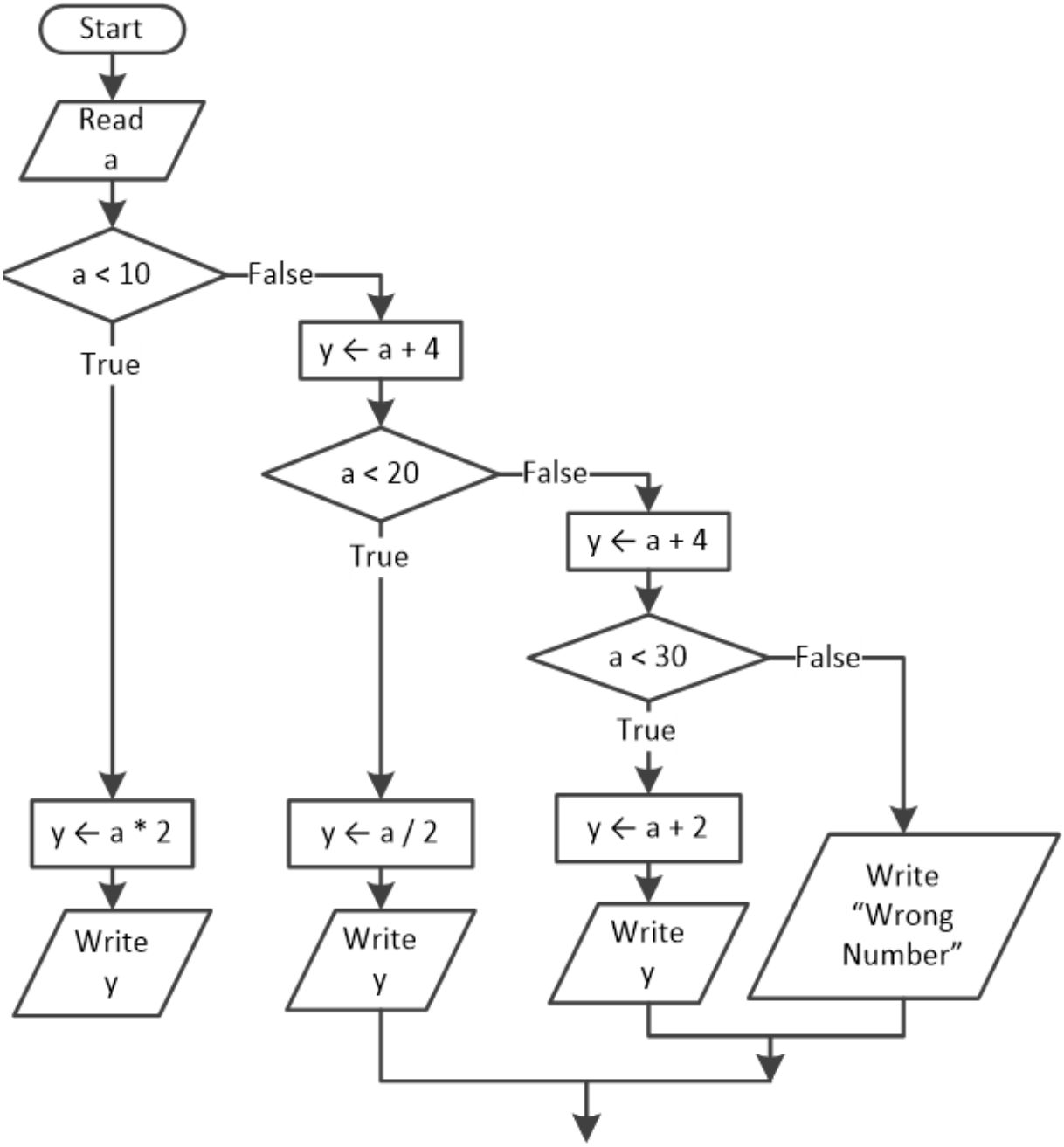
Let's say that you are in the middle of designing a flowchart (see the flowchart that follows), and you want to start closing all of its decision control structures.



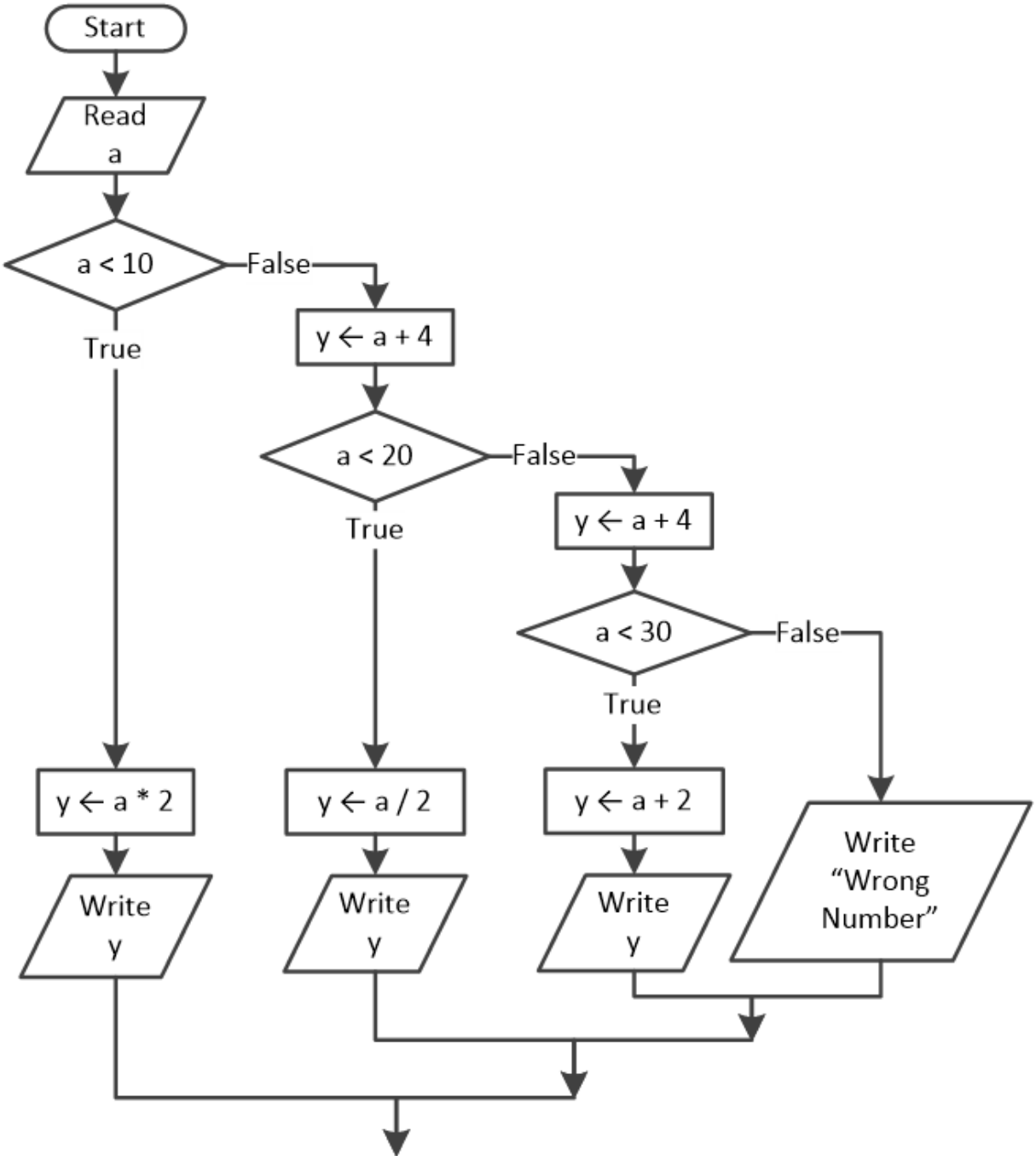
Just remember that the decision control structure that opens last must be the first one to close! In this example, the last decision control structure is the one that evaluates the expression `a < 30`. This is the first one that you need to close, as shown here.



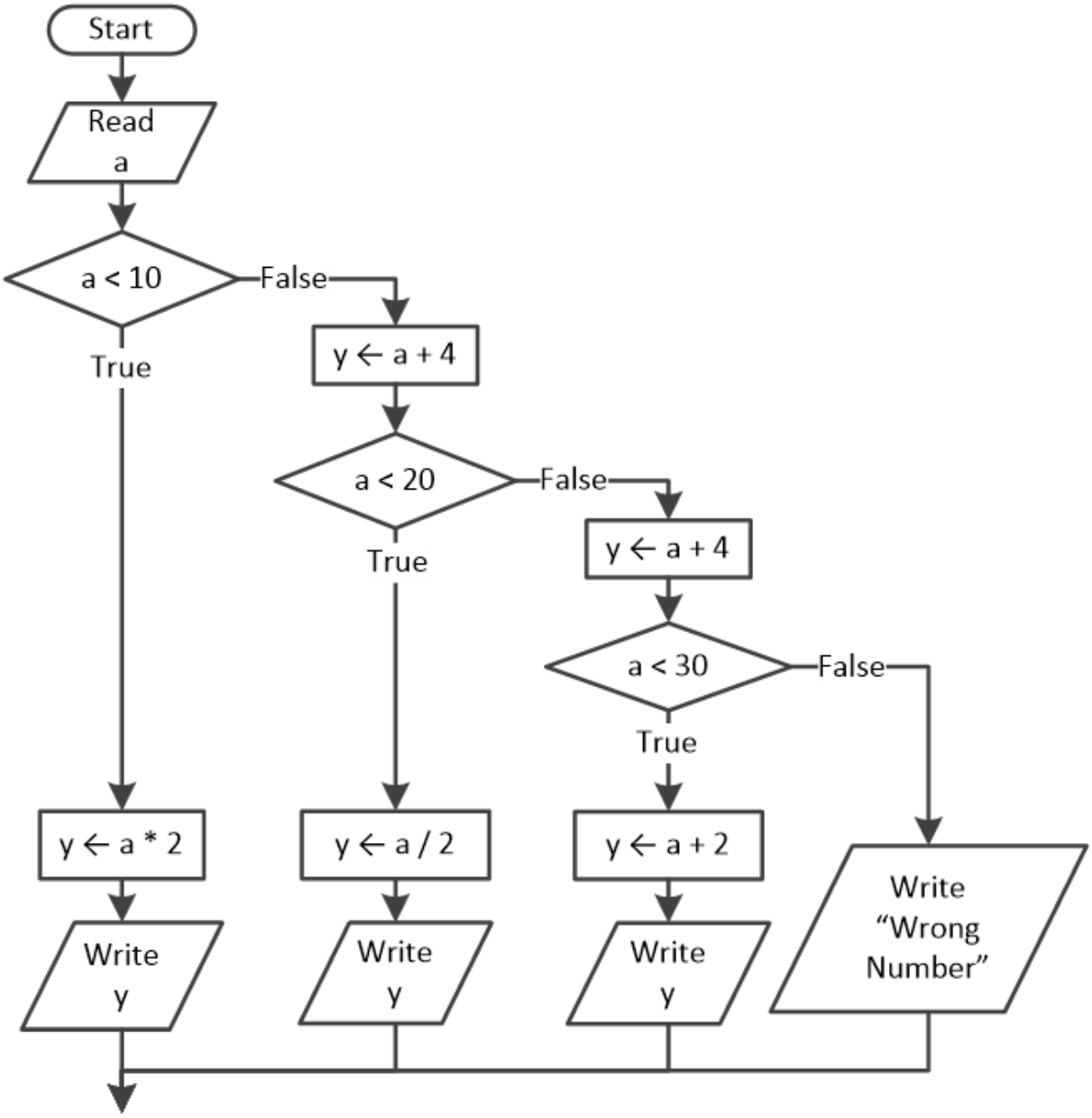
Next, you need to close the second to last decision control structure as shown here.



And finally, you need to close the third to last decision control structure as shown here.

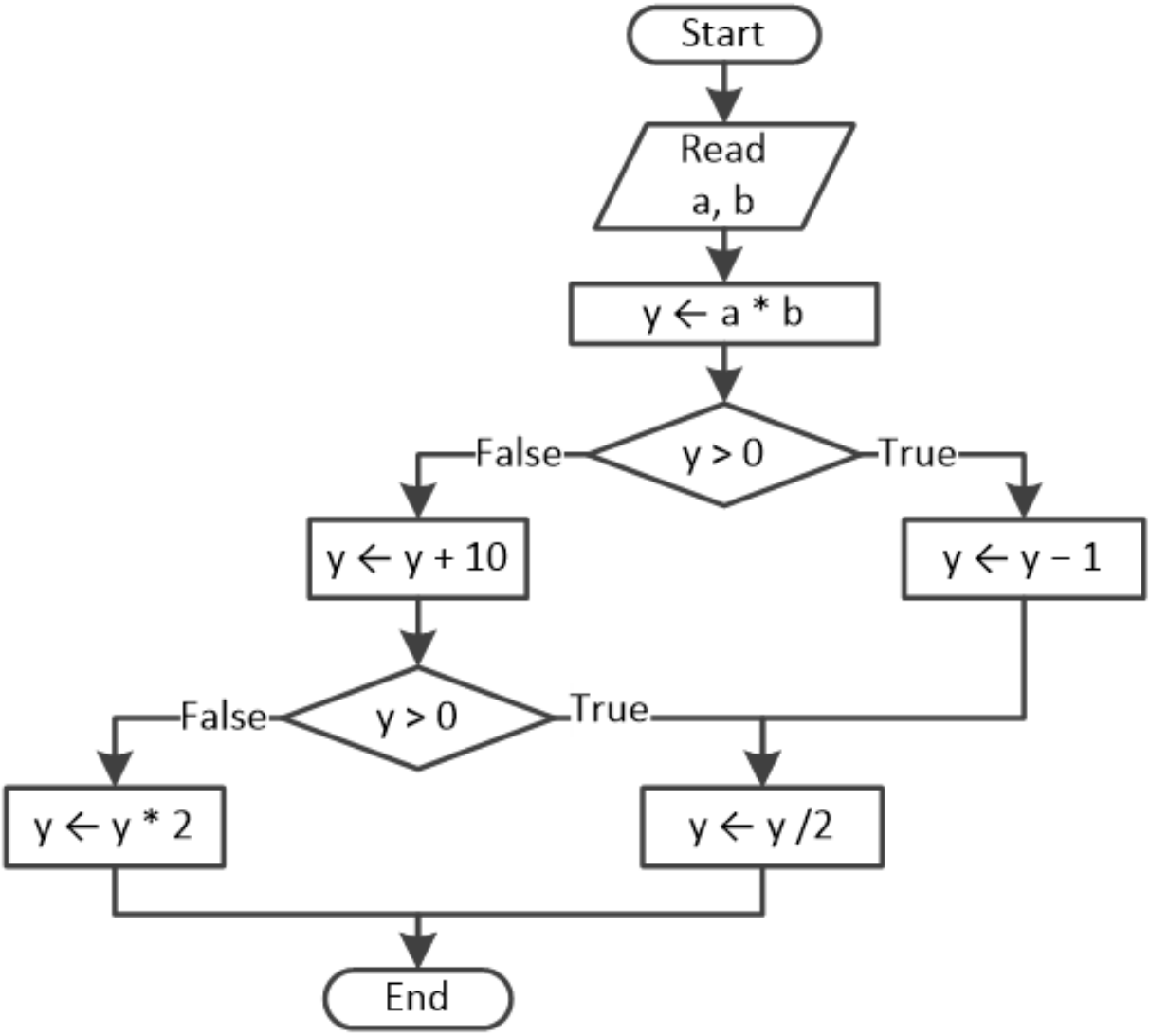


The last flowchart can be rearranged to become like the one shown here.



21.4 Converting Flowcharts to Java Programs

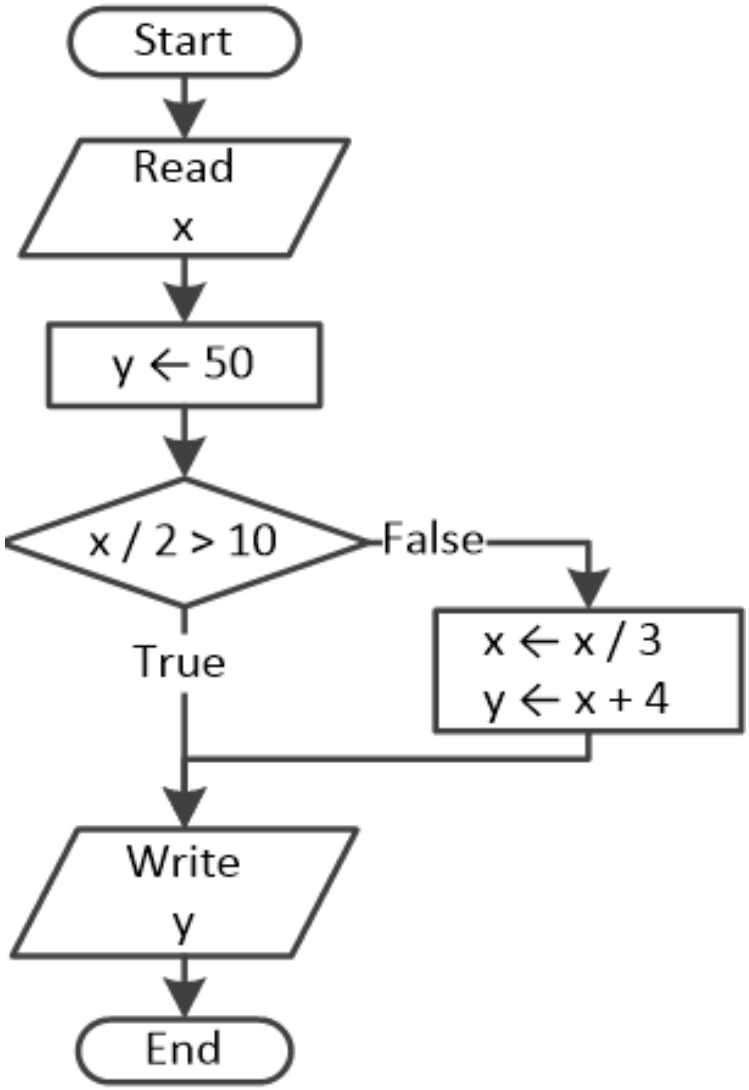
This conversion is not always an easy one. There are cases in which the flowchart designers follow no particular rules, so the initial flowchart may need some modifications before it can be converted into a Java program. An example of one such case is as follows.



As you can see, the decision control structures included in this flowchart fragment do not match any of the decision control structures that you have already learned. Thus, you have only one choice and this is to modify the flowchart by adding extra statements or removing existing ones until known decision control structures start to appear. Following are some exercises in which the initial flowchart does need modification.

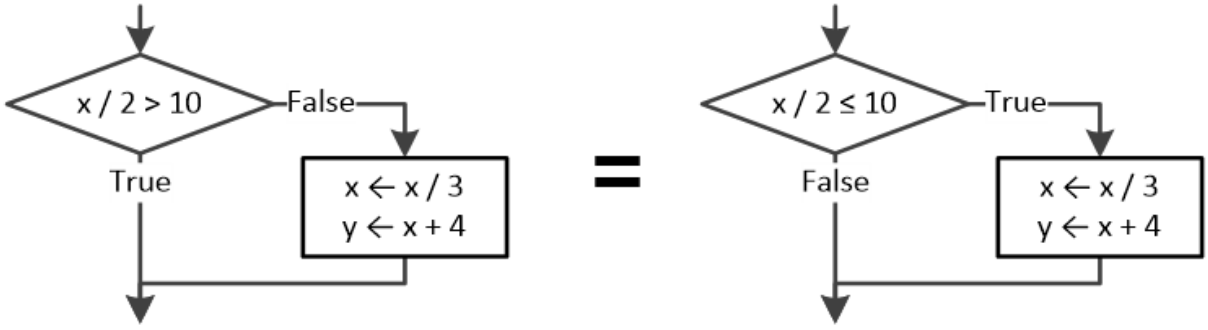
Exercise 21.4-1 Writing the Java Program

Write the Java program that corresponds to the following flowchart.

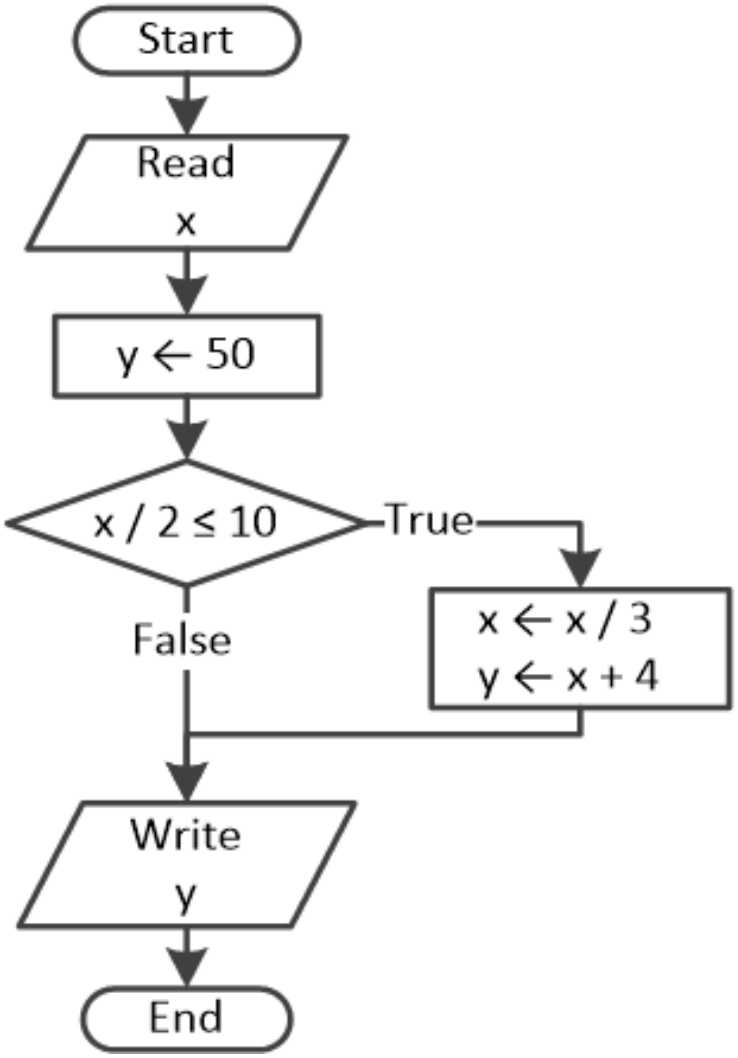


Solution This is quite easy. The only obstacle you must overcome is that the true and false paths are not quite in the right positions. You need to use the true path, and not the false path, to actually include the statements in the single-alternative decision structure.

It is possible to switch the two paths, but you also need to negate the corresponding Boolean expression. The following two flowchart fragments are equivalent.



Thus, the flowchart can be modified and look like this.



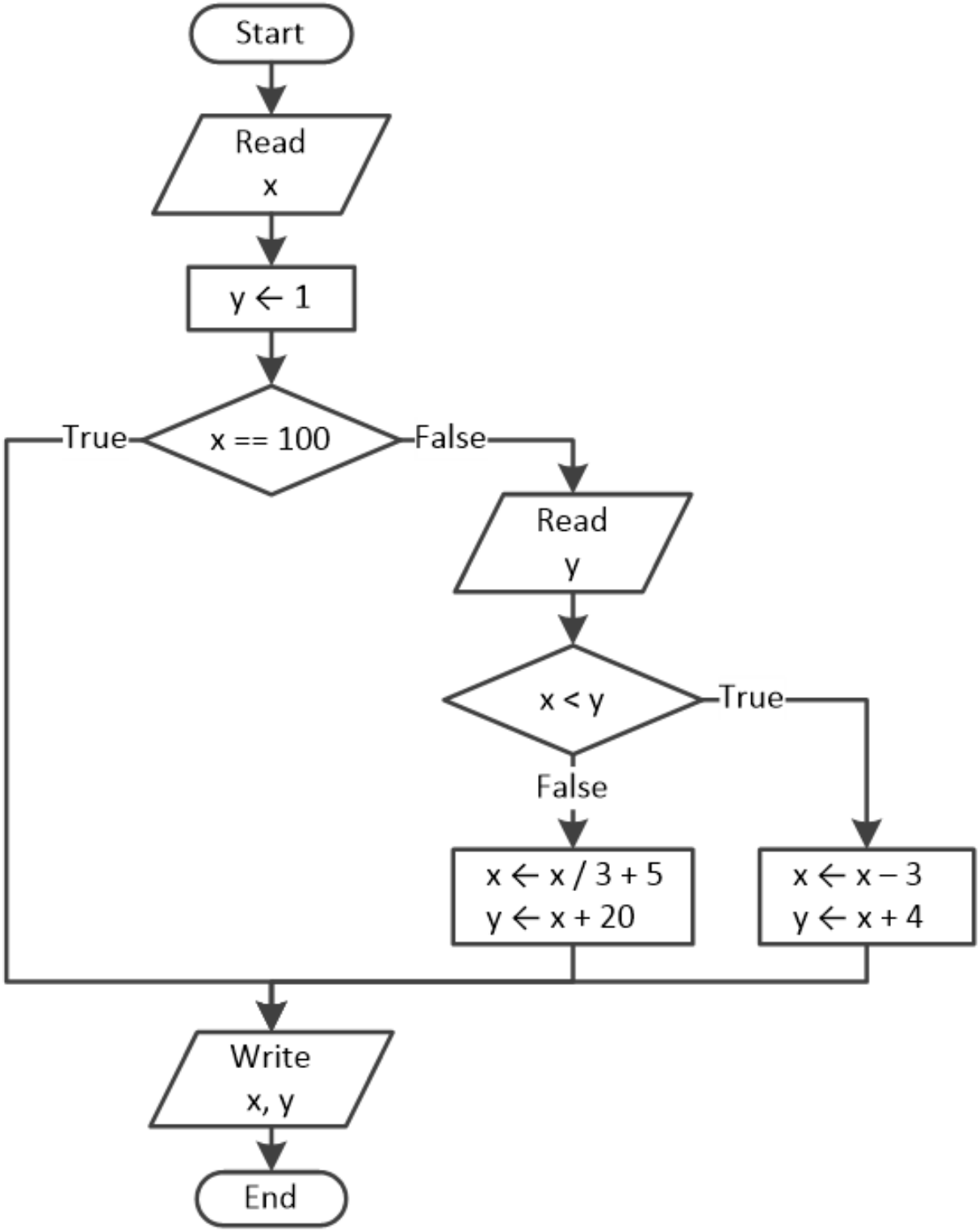
and the corresponding Java program is shown here.

```
public static void main(String[] args) throws Exception {
   double x, y;
   x = Double.parseDouble(cin.nextLine());
   y = 50; if (x / 2 <= 10) {
      x = x / 3;
```

```
    y = x + 4;
}
System.out.println(y); }
```

Exercise 21.4-2 Writing the Java Program

Write the Java program that corresponds to the following flowchart.

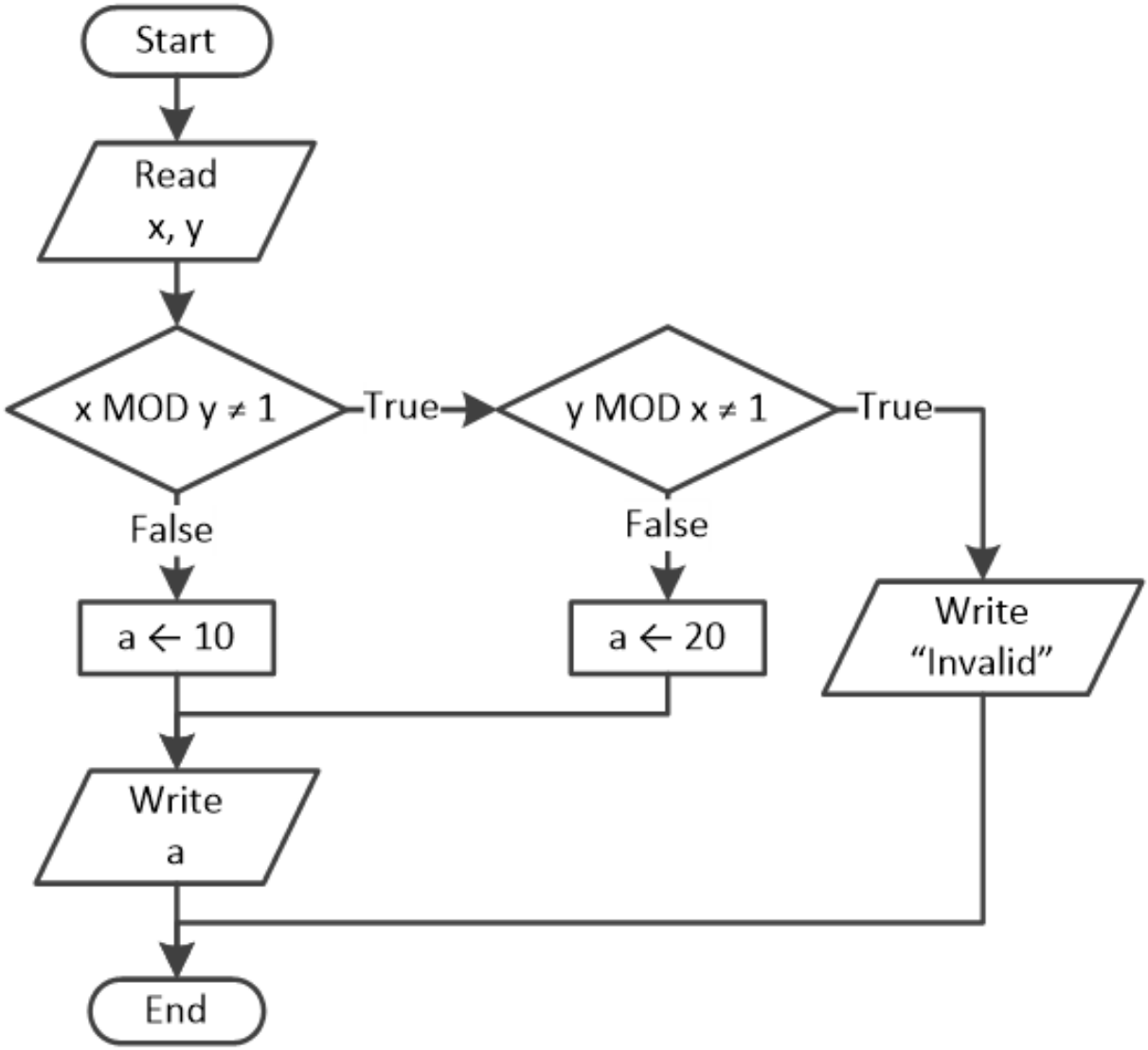


Solution In this exercise there is a dual-alternative decision structure nested within a single-alternative one. You just need to negate the Boolean expression `x == 100` ***and switch the true/false paths. The Java program is shown here.***

```
public static void main(String[] args) throws Exception {
   double x, y;
   x = Double.parseDouble(cin.nextLine());
   y = 1;
   if (x != 100) { //This is a single-alternative decision structure
   y = Double.parseDouble(cin.nextLine());
      if (x < y) { //This is a nested dual-alternative decision structure
         x = x - 3;
         y = x + 4;
      }
      else {
         x = x / 3 + 5;
         y = x + 20;
      }
   }
   System.out.println(x + " " + y); }
```

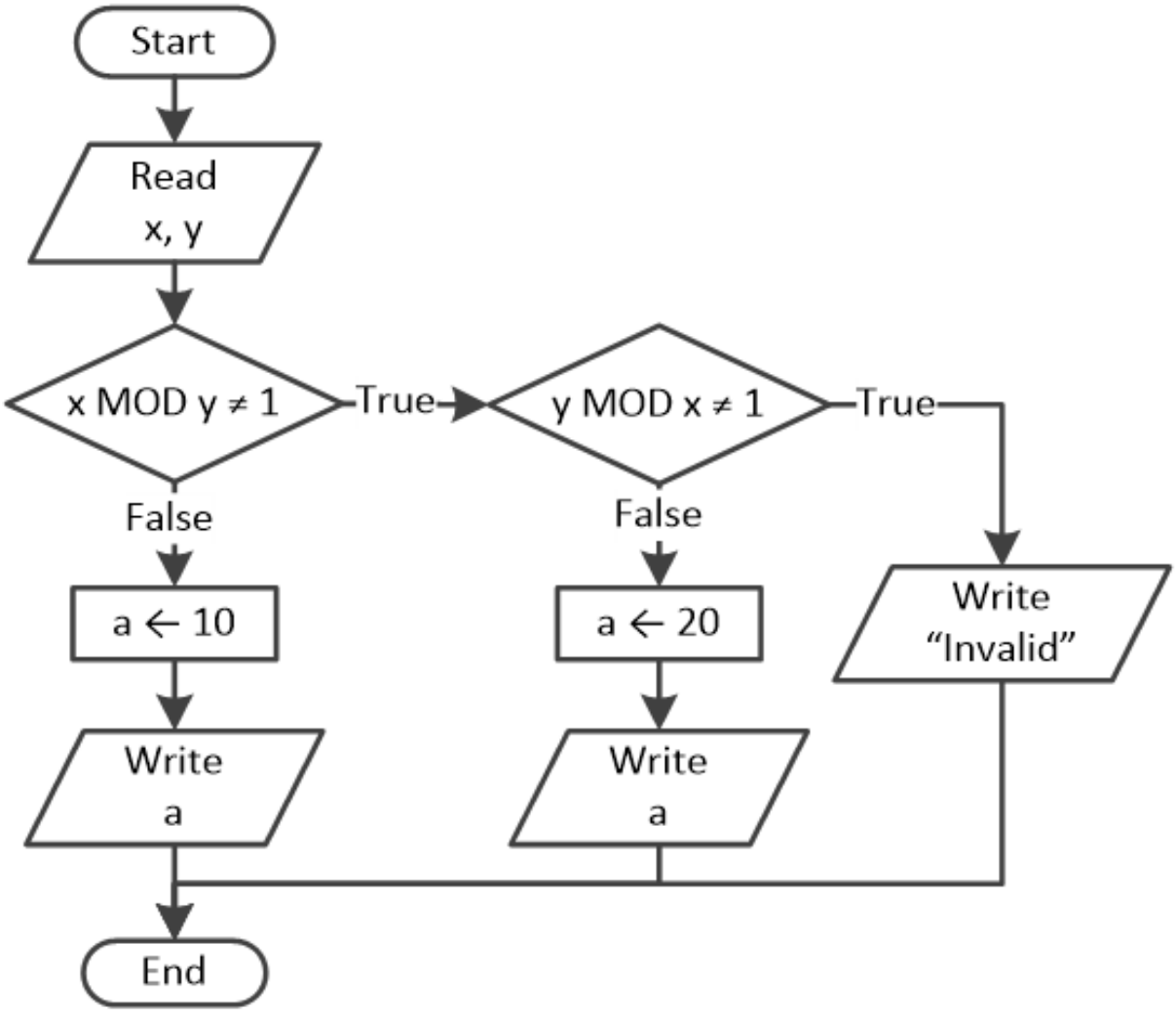
Exercise 21.4-3 Writing the Java Program

Write the Java program that corresponds to the following flowchart.



Solution In this flowchart, the decision control structures do not match any of the decision control structures that you learned. Thus, you must modify the flowchart by adding extra statements or removing existing ones until known decision control structures start to appear!

The obstacle you must overcome in this exercise is the decision control structure that evaluates the `y MOD x ≠ 1` Boolean expression. Note that when flow of execution follows the false path, it executes the statement `a ← 20` and then the statement `Write a` before it reaches the end of the algorithm. Thus, if you simply add a new statement, `Write a`, inside its false path you can keep the flow of execution intact. The following flowchart is equivalent to the initial one.

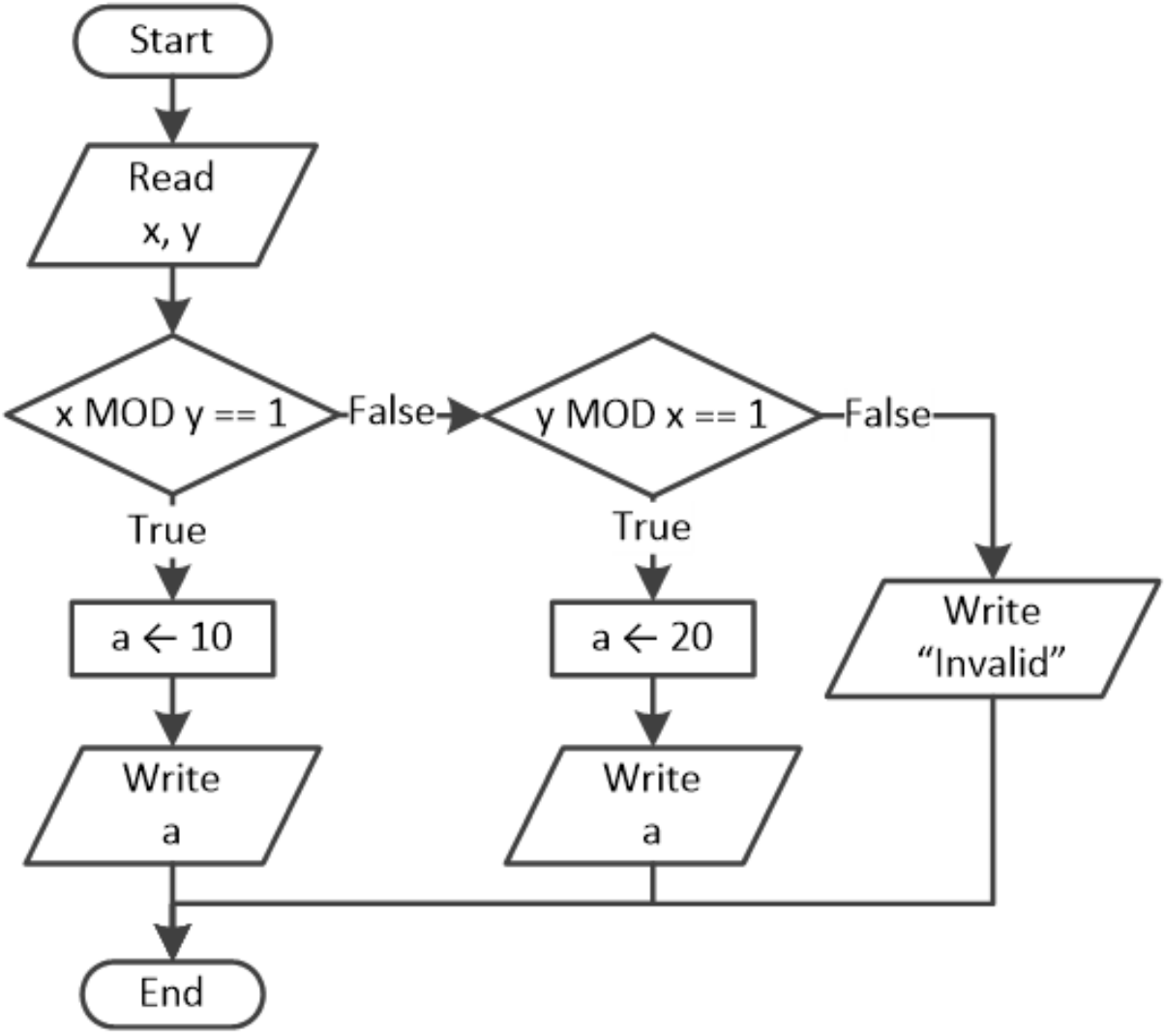


Now, the flowchart includes known decision control structures; that is, a dual-alternative decision structure nested within another dual-alternative one. The corresponding Java program is as follows.

```
public static void main(String[] args) throws Exception {
   int x, y, a;
   x = Integer.parseInt(cin.nextLine()); y = Integer.parseInt(cin.nextLine());
   if (x % y != 1) {
      if (y % x != 1) {
         System.out.println("Invalid");
      }
      else {
         a = 20;
         System.out.println(a);
      }
   }
   else {
```

```
      a = 10;
      System.out.println(a);
   }
}
```

However, there is something better that you can do! If you negate all Boolean expressions and also switch their true/false paths, you can have a multiple-alternative decision structure, which is more convenient in Java than nested decision control structures. The modified flowchart is shown here.



and the corresponding Java program is as follows.

```
public static void main(String[] args) throws Exception {
   int x, y, a;
   x = Integer.parseInt(cin.nextLine()); y = Integer.parseInt(cin.nextLine());
   if (x % y == 1) {
      a = 10;
```

```
      System.out.println(a);
   }
   else if (y % x == 1) {
      a = 20;
      System.out.println(a);
   }
   else {
      System.out.println("Invalid");
   }
}
```

21.5 Review Exercises

Complete the following exercises.

1) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a;
   a = Integer.parseInt(cin.nextLine());
   if (a % 10 == 0) {
      a++;
      System.out.println("Message #1");
   }
   if (a % 3 == 1) {
      a += 5;
      System.out.println("Message #2");
   }
   if (a % 3 == 2) {
      a += 10;
      System.out.println("Message #3");
   }
   System.out.println(a); }
```

2) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a;
   a = Integer.parseInt(cin.nextLine());
   if (a % 10 == 0) {
      a++;
      System.out.println("Message #1");
   }
   if (a % 3 == 1) {
      a += 5;
      System.out.println("Message #2");
   }
   else {
```

```
      a += 7;
   }
   System.out.println(a); }
```

3) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   double a, y, b;
   a = Double.parseDouble(cin.nextLine());
   if (a < 0) {
      y = a * 2;
      if (y > 0)
         y +=2;
      else if (y == 0)
         y *= 6;
      else
         y /= 7;
   }
   else if (a < 22)
   y = a / 3;
   else if (a < 32)
   y = a - 7;
   else {
      b = Double.parseDouble(cin.nextLine());
      y = a - b;
   }
   System.out.println(y); }
```

4) Design the flowchart that corresponds to the following code fragment given in general form.

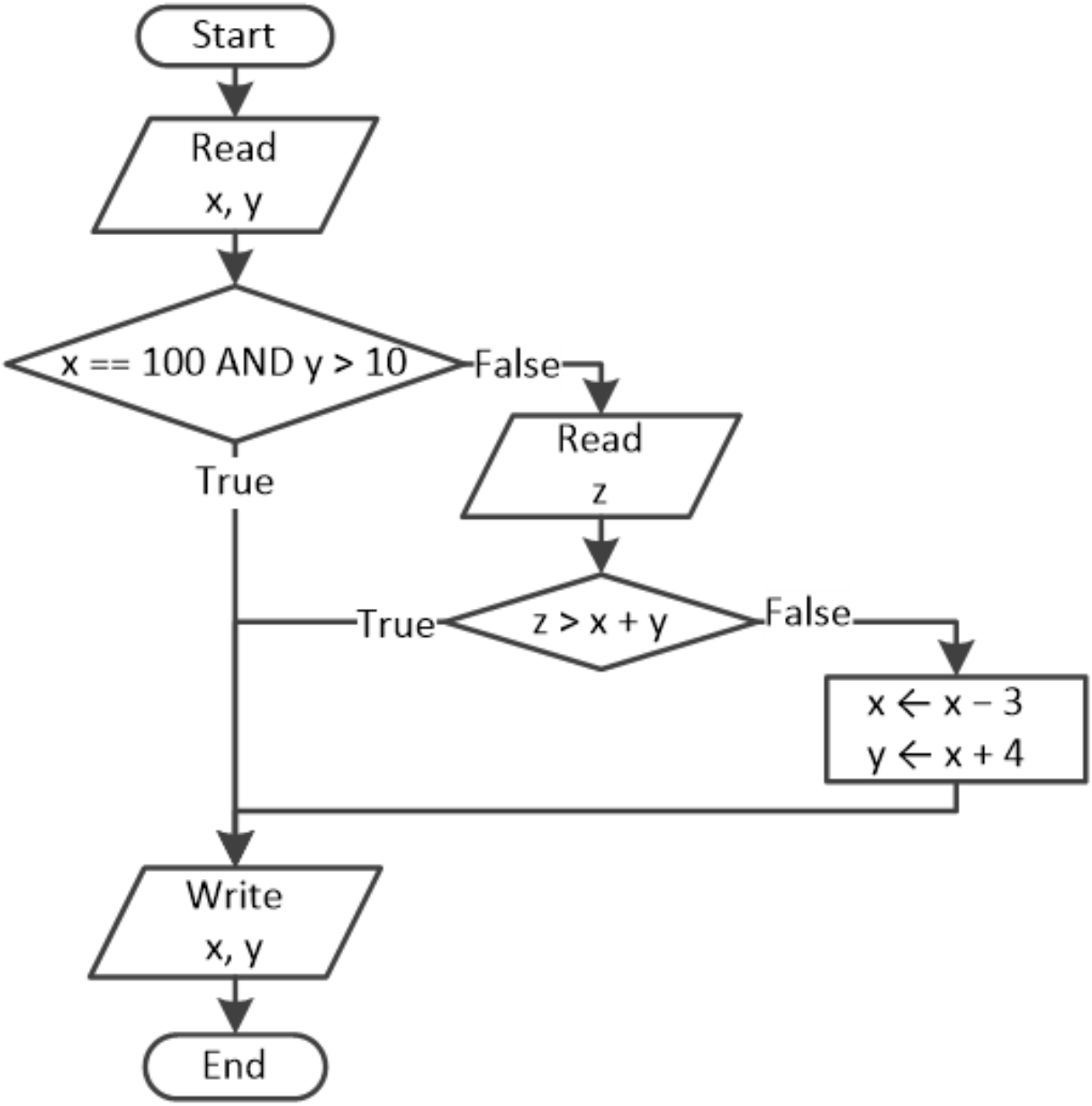
```
if (Boolean_Expression_A) {
   if (Boolean_Expression_B) {
      A statement or block of statements B1
   }
   else {
      A statement or block of statements B2
   }
   A statement or block of statements A1
}
else {
   A statement or block of statements A2
   if (Boolean_Expression_C) {
```

```
        A statement or block of statements C1
    }
    else if (Boolean_Expression_D) {
        A statement or block of statements D1
    }
    else {
        A statement or block of statements E1
    }
    A statement or block of statements A3
}
```

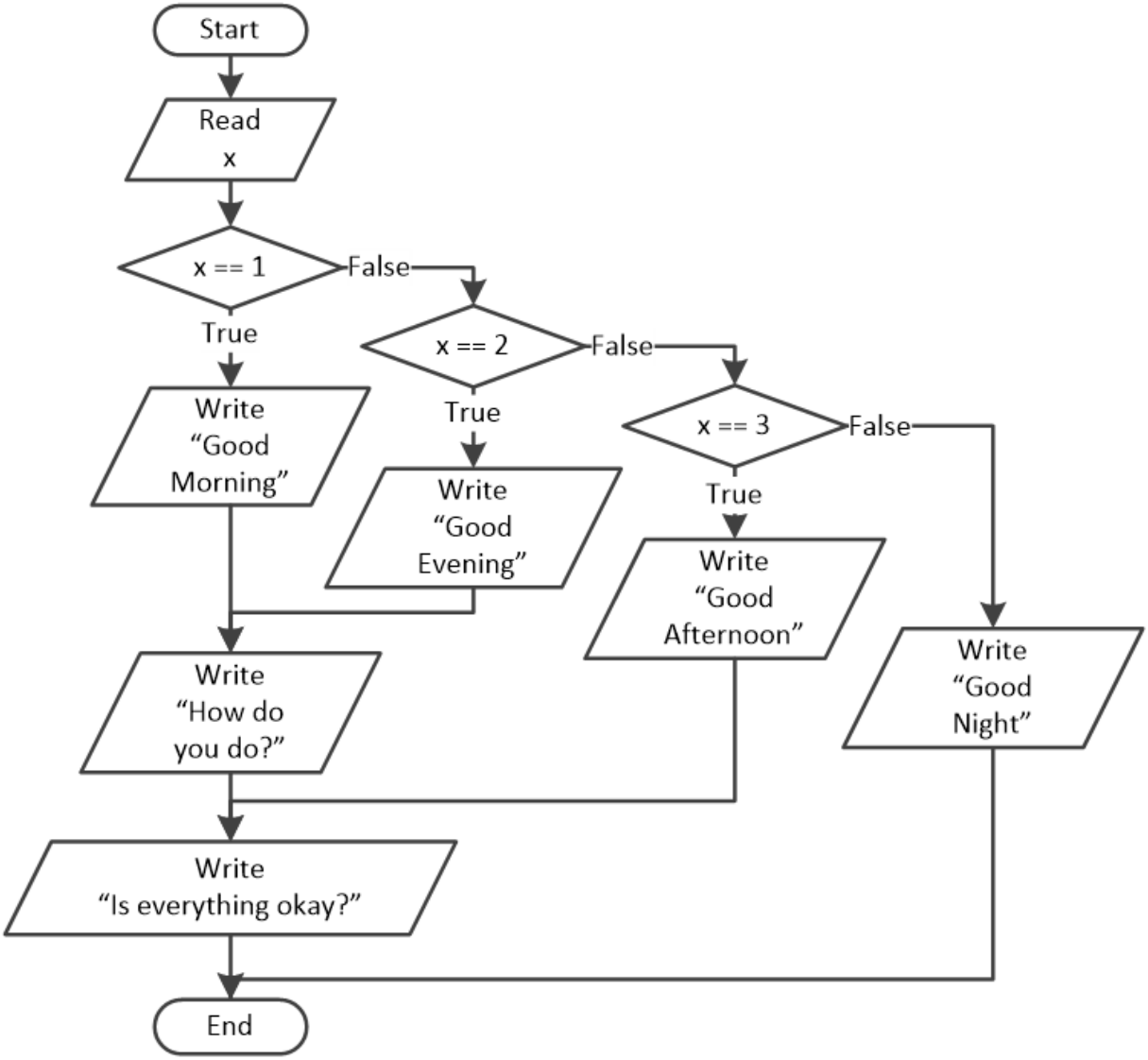
5) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a; double y, b;
   a = Integer.parseInt(cin.nextLine()); y = 0;
   switch (a) {
      case 1:
         y = a * 2;
         break;
      case 2:
         y = a - 3;
         break;
      case 3:
         y = a + 3;
         if (y % 2 == 1)
            y += 2;
         else if (y == 0)
            y *= 6;
         else
            y /= 7;
         break;
      case 4:
         b = Double.parseDouble(cin.nextLine());
         y = a + b + 2;
         break;
   }
   System.out.println(y); }
```

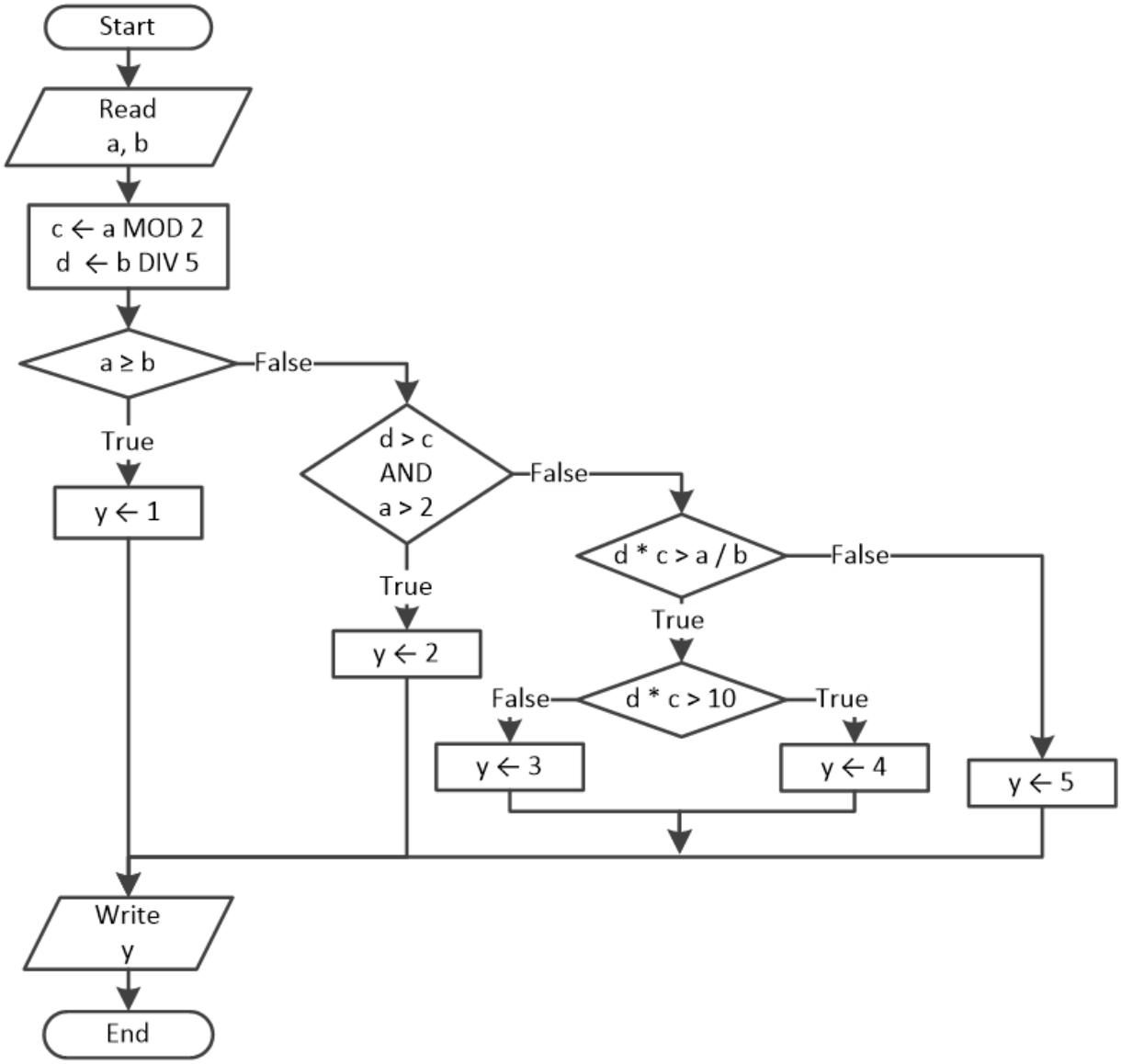
6) Write the Java program that corresponds to the following flowchart.



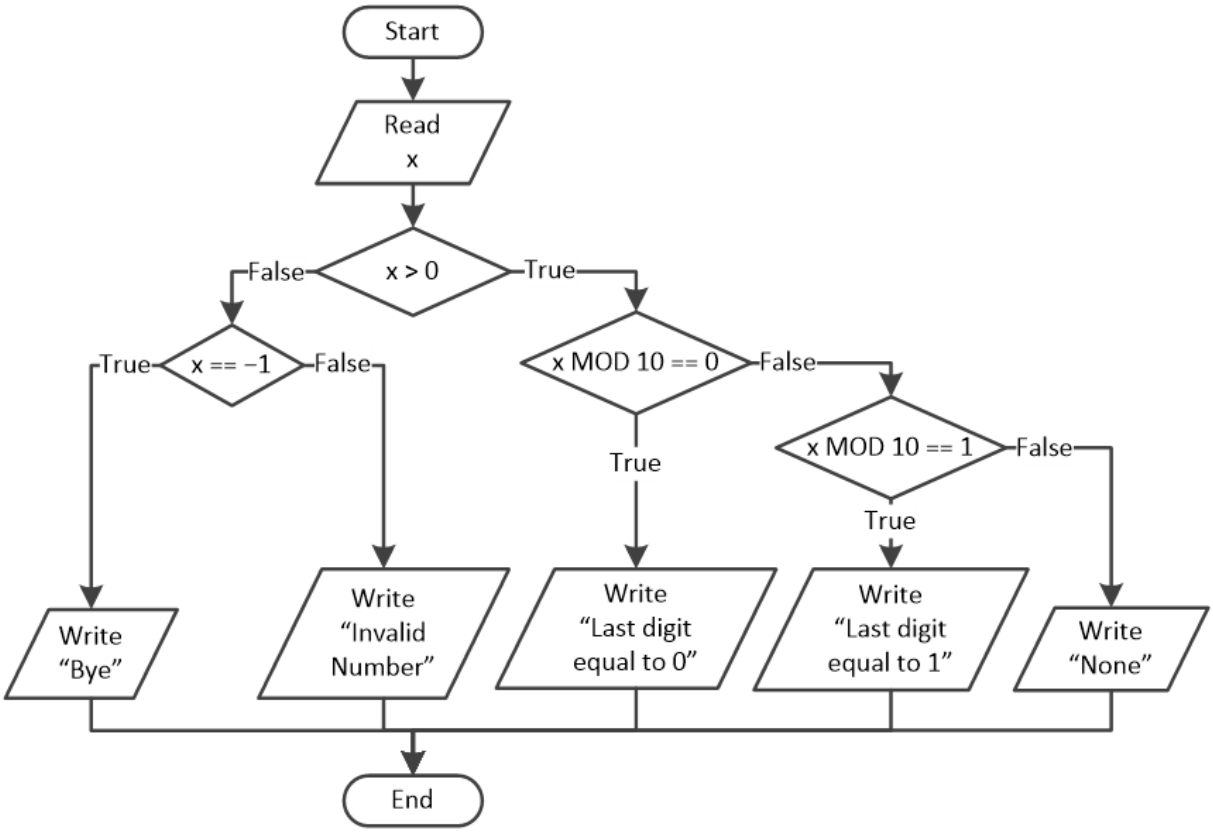
7) Write the Java program that corresponds to the following flowchart.



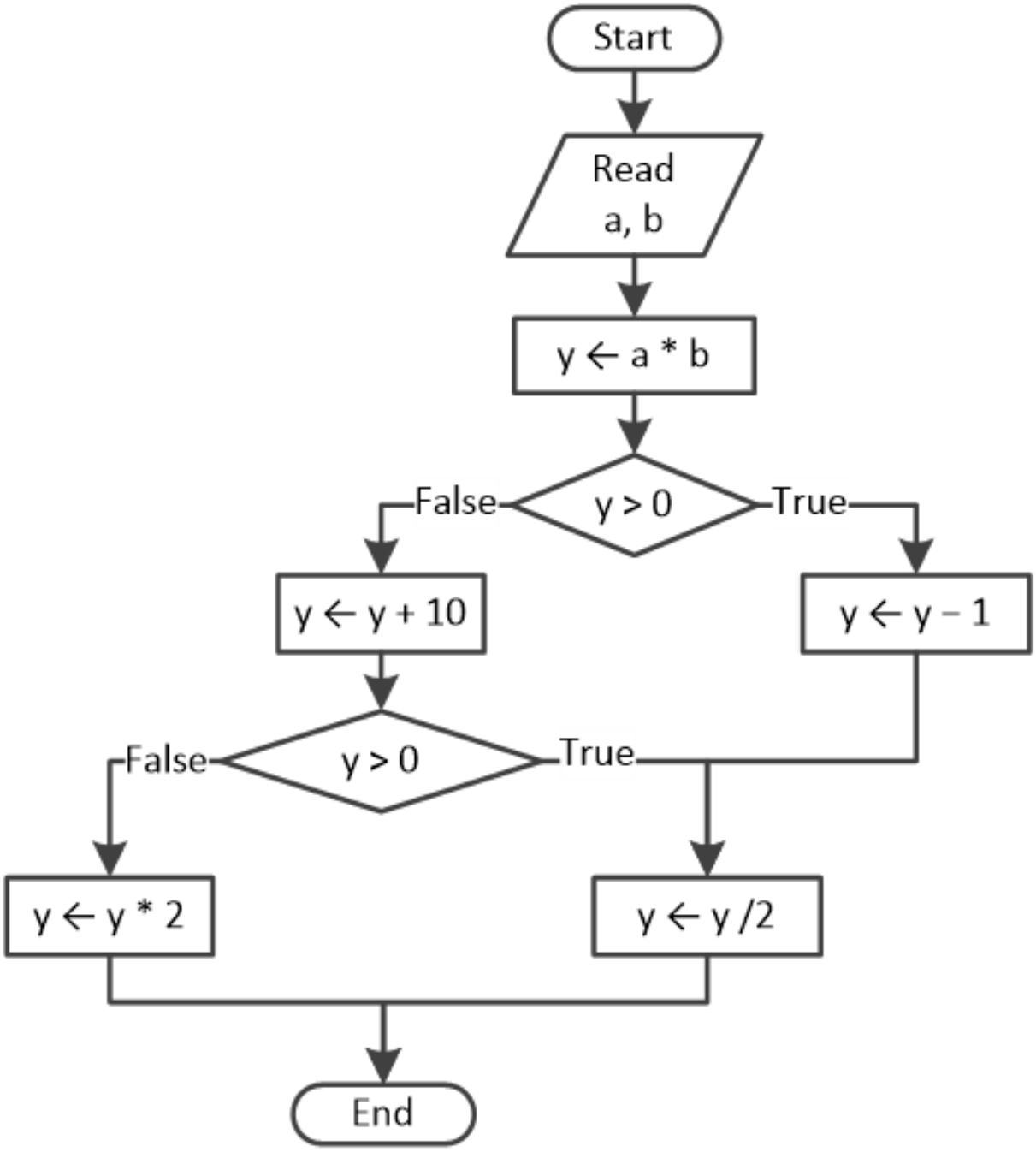
8) Write the Java program that corresponds to the following flowchart.



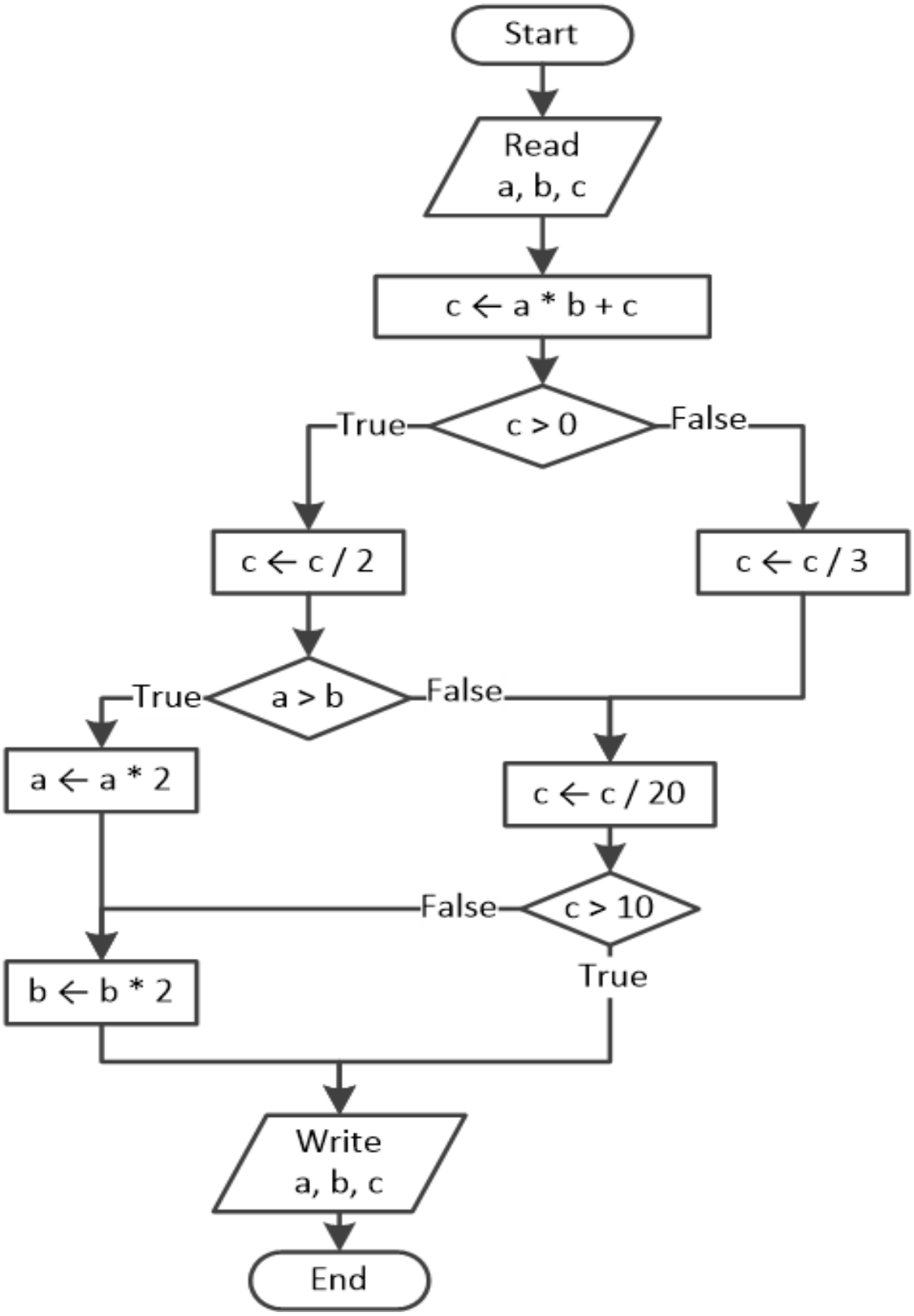
9) Write the Java program that corresponds to the following flowchart.



10) Write the Java program that corresponds to the following flowchart.



11) Write the Java program that corresponds to the following flowchart.


Start
Read
a, b, c
c ← a * b + c
c > 0
True
False
c ← c / 2
c ← c / 3
a > b
True
False
a ← a * 2
c ← c / 20
c > 10
False
True
b ← b * 2
Write
a, b, c
End

Chapter 22
Tips and Tricks with Decision Control Structures

22.1 Introduction

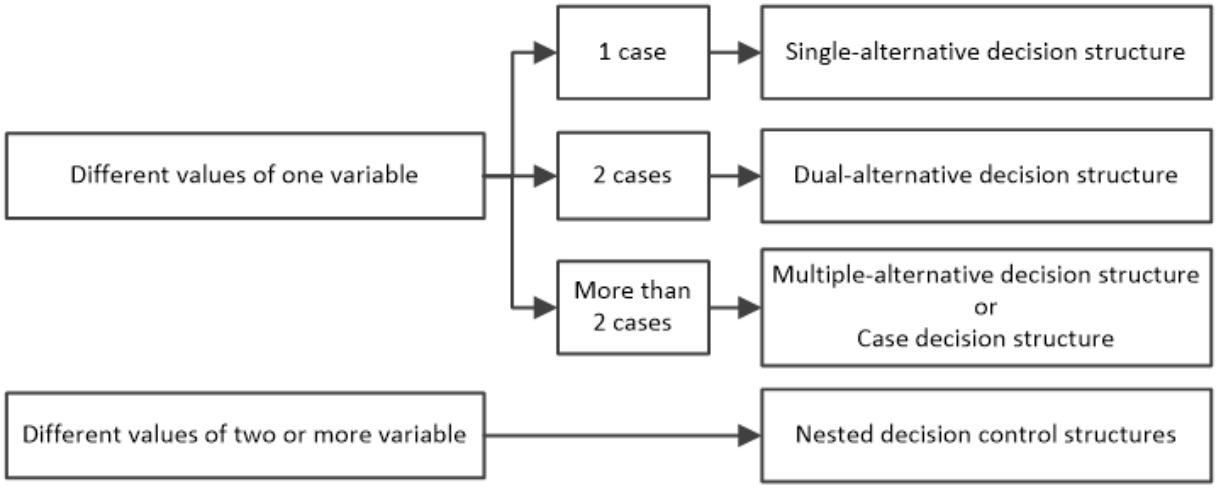
This chapter is dedicated to teaching you some useful tips and tricks that can help you write “better” code. You should always keep them in mind when you design your own algorithms, or even your own Java programs.

These tips and tricks can help you increase your code's readability and help make the code shorter or even faster. Of course there is no single perfect methodology because on one occasion the use of a specific tip or trick may help, but on another occasion the same tip or trick may have exactly the opposite result. Most of the time, code optimization is a matter of programming experience.

Smaller algorithms are not always the best solution to a given problem. In order to solve a specific problem, you might write a very short algorithm that unfortunately proves to consume a lot of CPU time. On the other hand, you may solve the same problem with another algorithm which, even though it seems longer, calculates the result much faster.

22.2 Choosing a Decision Control Structure

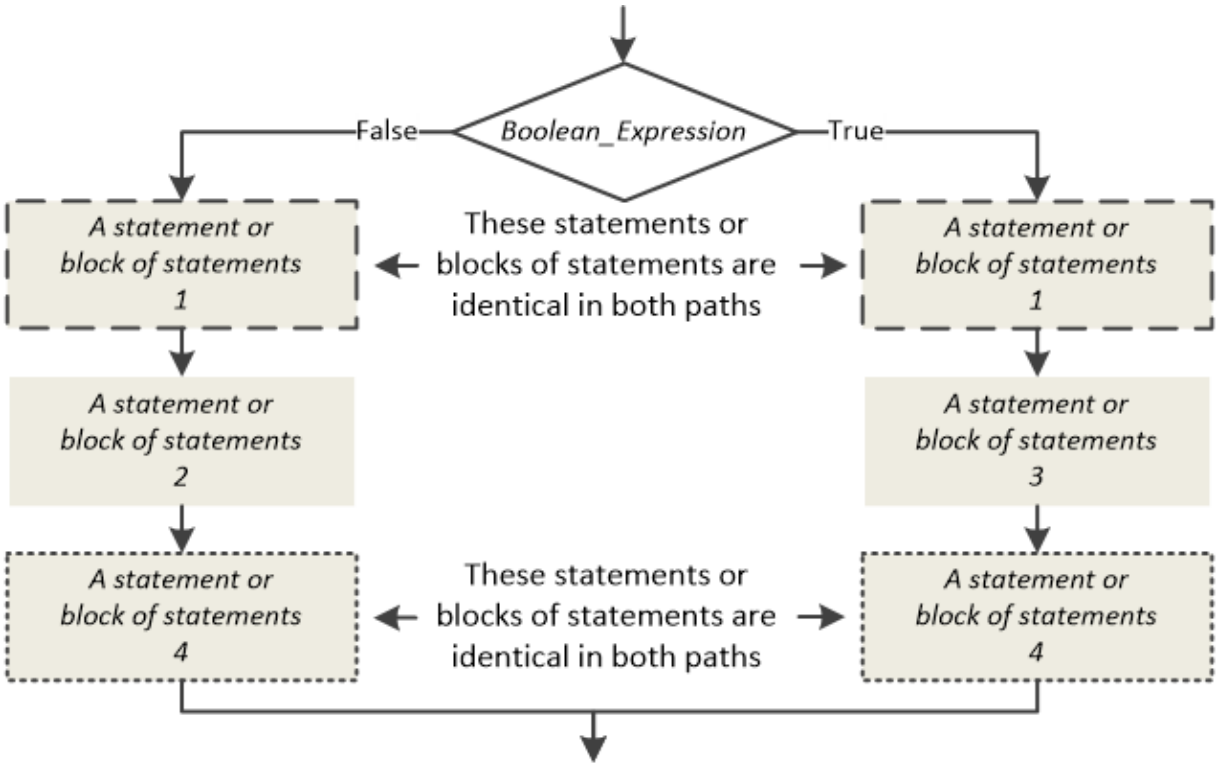
The following diagram can help you decide which decision control structure is a better choice for a given problem depending on the number of variables checked.


Different values of one variable
1 case
Single-alternative decision structure
2 cases
Dual-alternative decision structure
More than 2 cases
Multiple-alternative decision structure or Case decision structure
Different values of two or more variable
Nested decision control structures

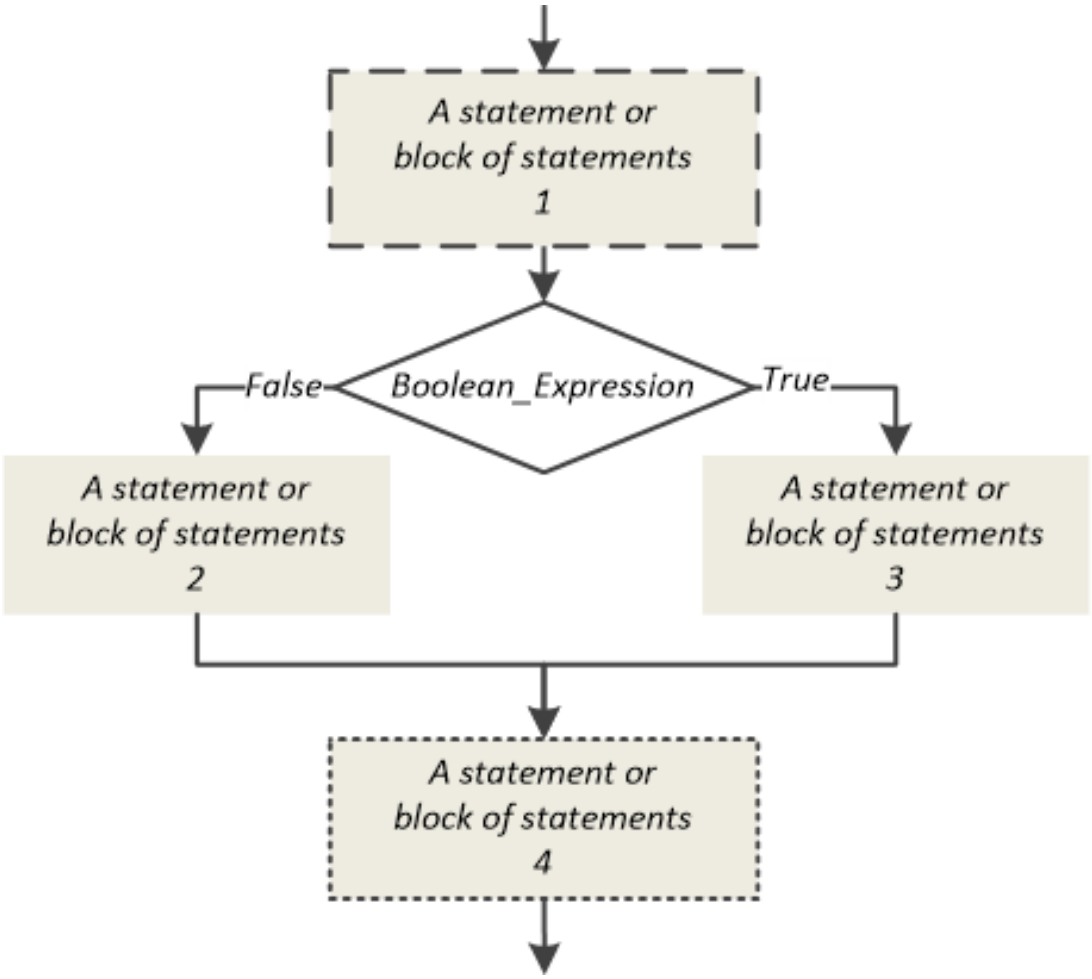
✎ *This diagram recommends the best option, not the only option. For example, when there are more than two cases for one variable, it is not wrong to use a nested decision control structure instead. The proposed multiple-alternative decision structure and the proposed case decision structure, though, are more convenient.*

22.3 Streamlining the Decision Control Structure

Look carefully at the following flowchart fragment given in general form.


Boolean_Expression
False
True
A statement or block of statements 1
These statements or blocks of statements are identical in both paths
A statement or block of statements 1
A statement or block of statements 2
A statement or block of statements 3
A statement or block of statements 4
These statements or blocks of statements are identical in both paths
A statement or block of statements 4

As you can see, two identical statements or blocks of statements exist at the beginning and two other identical statements or blocks of statements exist at the end of both paths of the dual-alternative decision structure. This means that, regardless of the result of *Boolean_Expression*, these statements are executed either way. Thus, you can simply move them outside and (respectively) right before and right after the dual-alternative decision structure, as shown in this equivalent structure.


A statement or block of statements 1
Boolean_Expression
False
True
A statement or block of statements 2
A statement or block of statements 3
A statement or block of statements 4

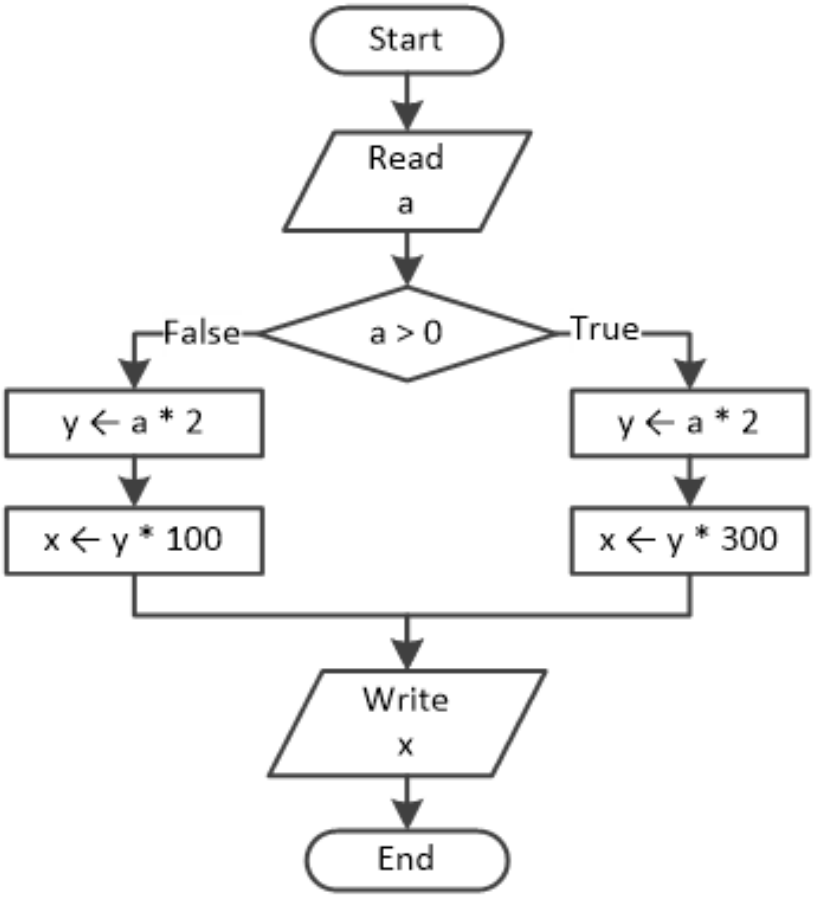
✎ *The same tip can be applied to any decision control structure, as long as an identical statement or block of statements exists in all paths.*

✎ *There are cases where this tip cannot be applied. For instance, you cannot move a statement (or block of statements) right before the decision control structure if this statement affects the Boolean expression of the structure.*

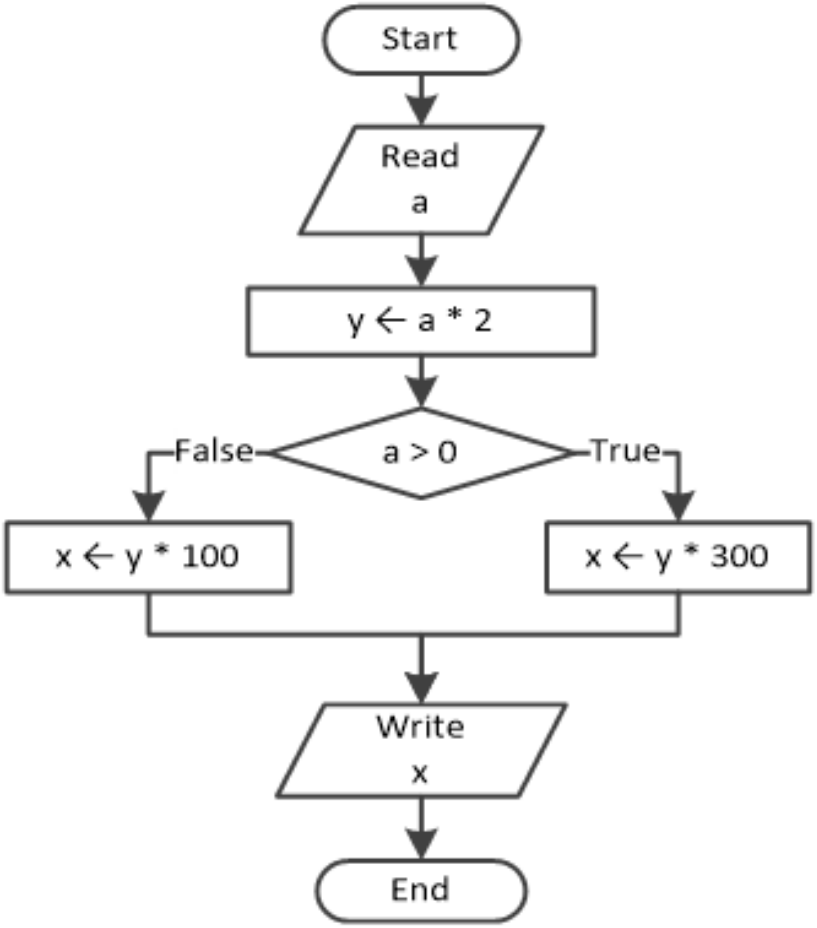
Are you still confused? Next, you will find some exercises that can help you to understand better.

Exercise 22.3-1 "Shrinking" the Algorithm

Redesign the following flowchart using fewer statements.



Solution As you can see, the statement y ← a * 2 exists in both paths of the dual-alternative decision structure. This means that, regardless of the result of the Boolean expression, this statement is executed either way. Therefore, you can simply move the statement outside and right before the dual-alternative decision structure, as follows.



Exercise 22.3-2 "Shrinking" the Java Program

Rewrite the following Java program using fewer statements.

```
public static void main(String[] args) throws Exception {
   int a, y;
   a = Integer.parseInt(cin.nextLine());
   if (a > 0) {
      y = a * 4;
      System.out.println(y);
   }
   else {
      y = a * 3;
      System.out.println(y);
   }
}
```

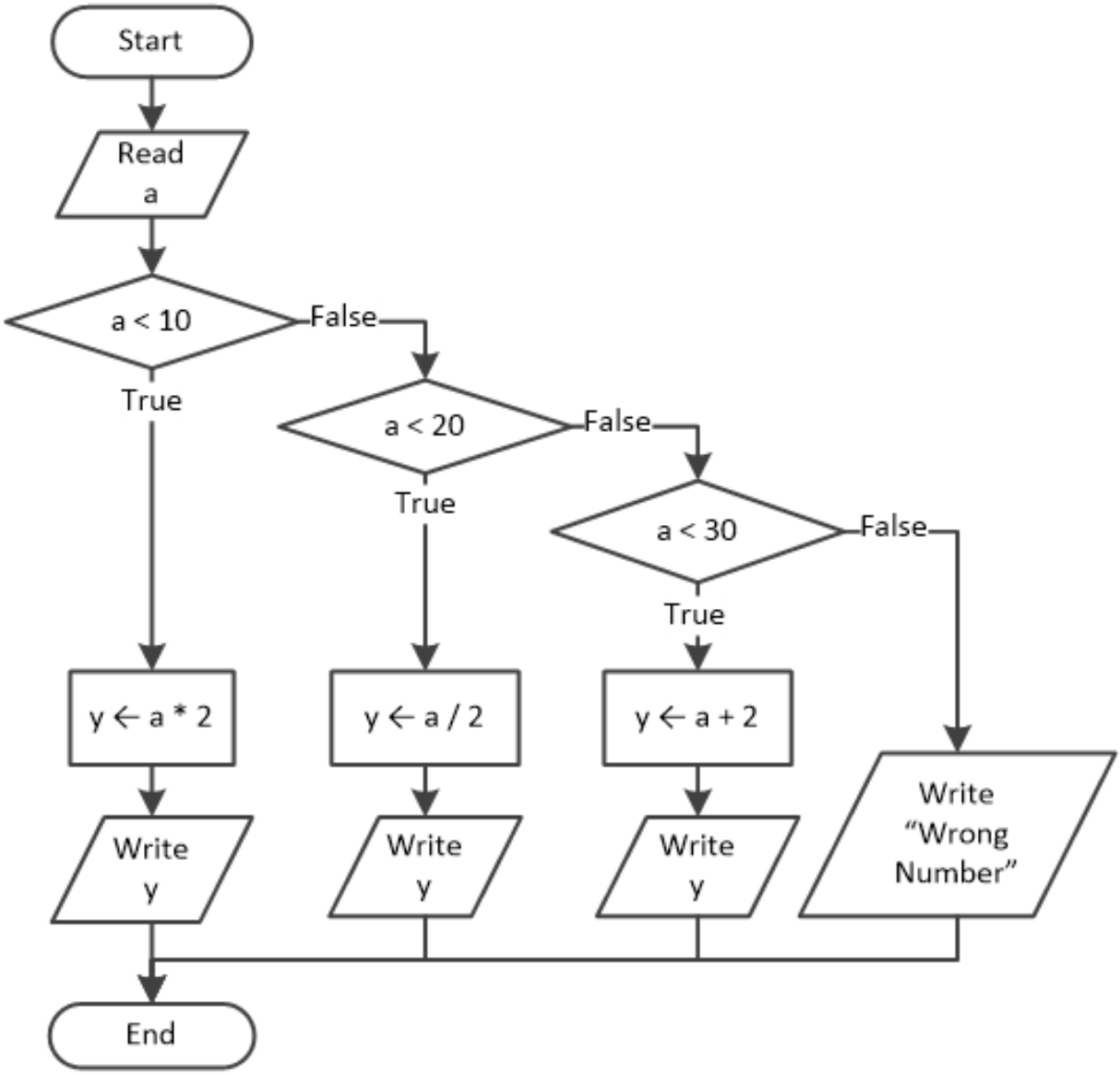
Solution As you can see, the statement `System.out.println(y)` exists in both paths of the dual-alternative decision structure. This means that, regardless of the result of the Boolean expression, this statement is executed

either way. Therefore, you can simply move the statement outside and right after the dual-alternative decision structure, as shown here.

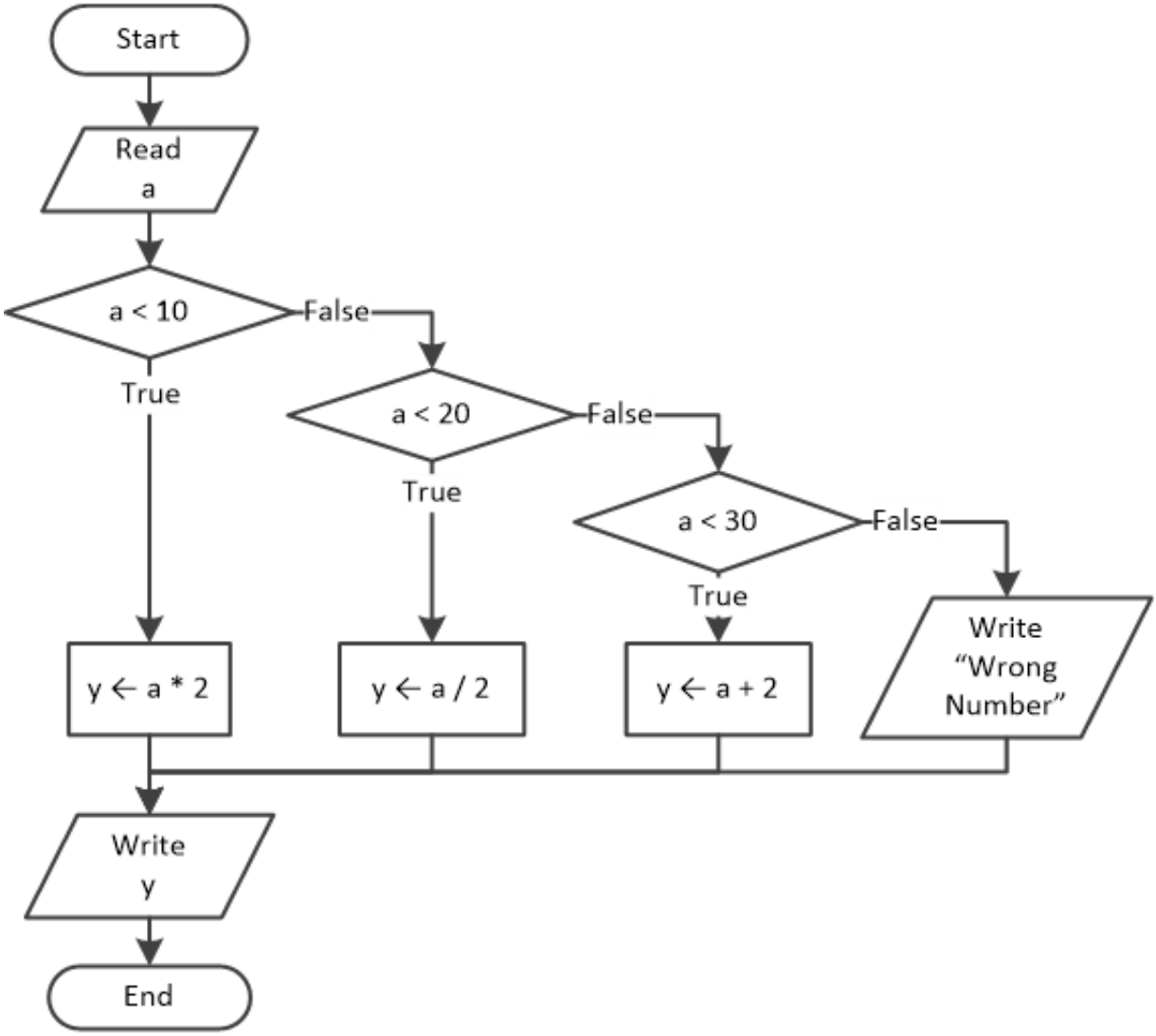
```
public static void main(String[] args) throws Exception {
   int a, y;
   a = Integer.parseInt(cin.nextLine());
   if (a > 0) {
      y = a * 4;
   }
   else {
      y = a * 3;
   }
   System.out.println(y); }
```

Exercise 22.3-3 “Shrinking” the Algorithm

Redesign the following flowchart using fewer statements and then write the corresponding Java program.



Solution If you try to move the `Write y` statement outside of the multiple-alternative decision structure, the resulting flowchart that follows is definitely not equivalent to the initial one.



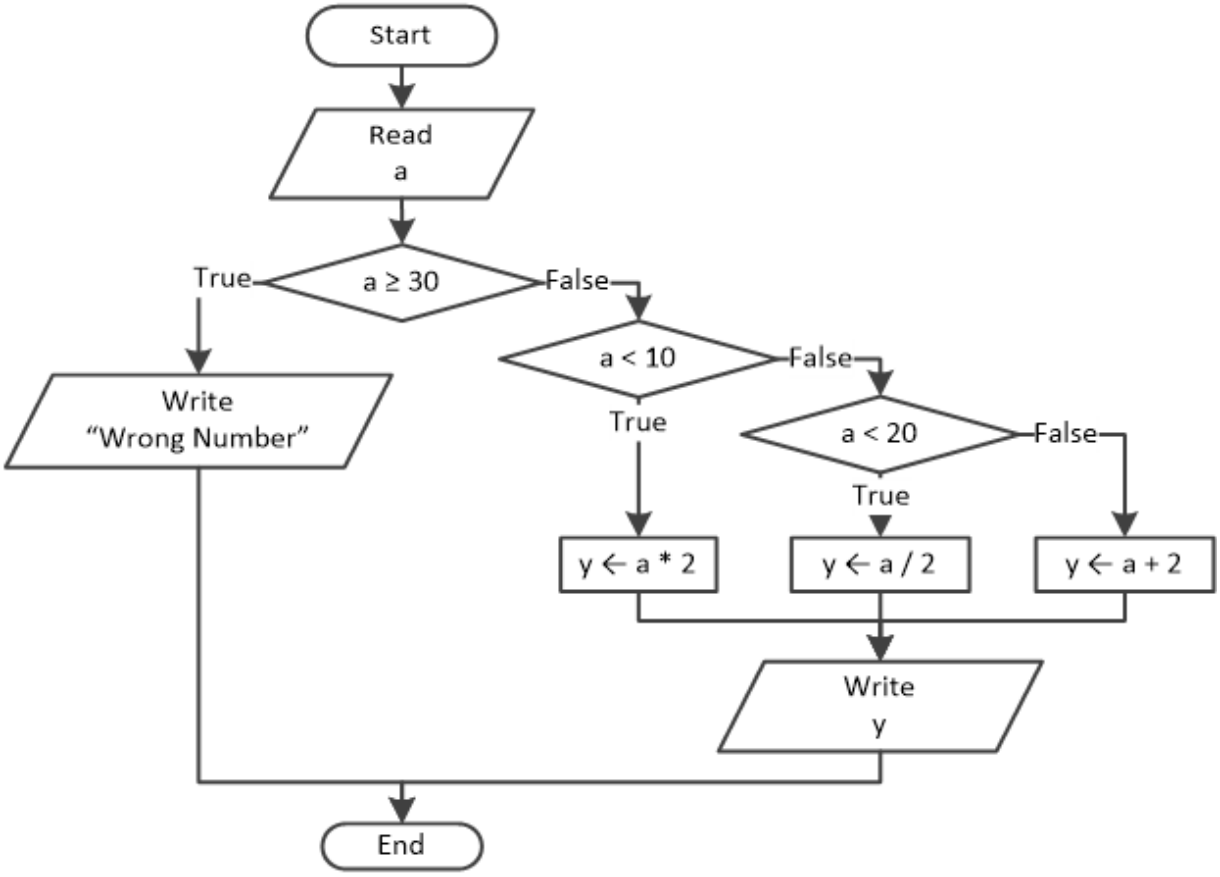
This is because of the last path on the right side which, in the initial flowchart, didn't include the `Write y` statement.

Examine both flowcharts to see whether they produce the same result. For example, suppose a user enters a wrong number. In both flowcharts, the flow of execution goes to the `Write "Wrong Number"` statement. After that, the initial flowchart executes no other statements whereas, the second flowchart executes an extra `Write y` statement.

You cannot move a statement or block of statements outside of a decision control structure if it does not exist in all paths.

You may now wonder whether there is any other way to move the `Write y` statement outside of the multiple-alternative decision structure. The answer is “yes”, but you need to slightly rearrange the flowchart. You need to

completely remove the last path on the right and use a brand new decision control structure in the beginning to check whether or not the user-provided number is wrong. One possible solution is shown here.



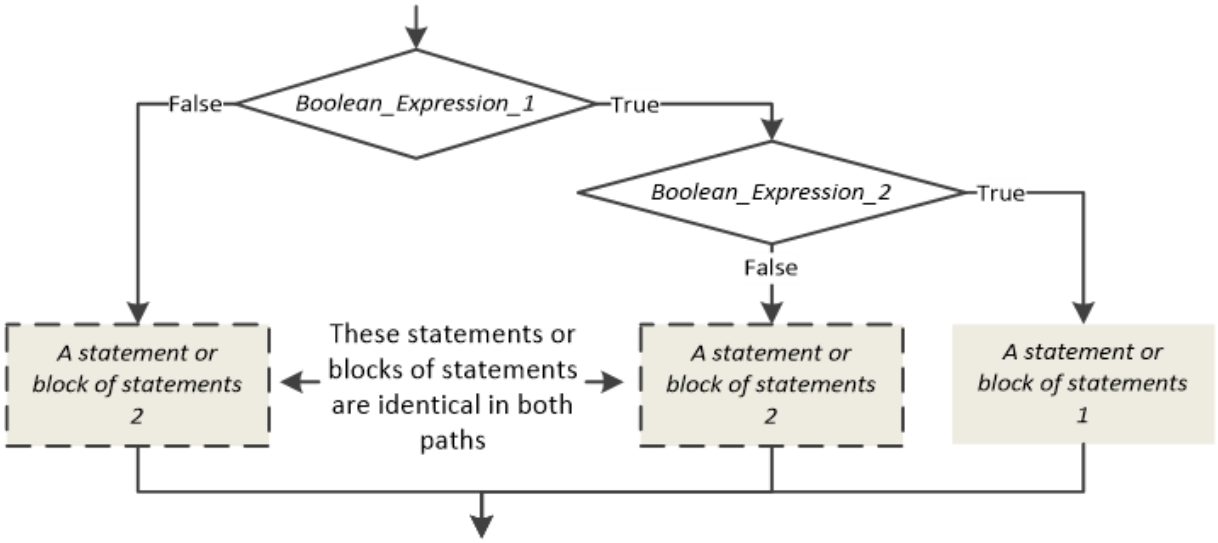
and the Java program is

```
public static void main(String[] args) throws Exception {
    double a, y;
    a = Double.parseDouble(cin.nextLine());
    if (a >= 30) {
        System.out.println("Wrong Number");
    }
    else {
        if (a < 10) {
            y = a * 2;
        }
        else if (a < 20) {
            y = a / 2;
        }
        else {
```

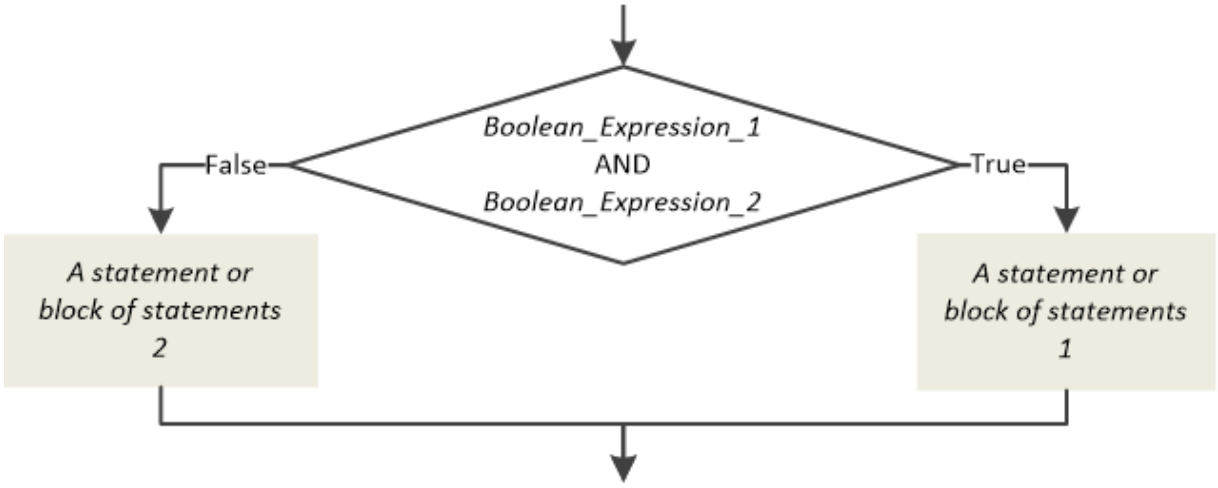
```
            y = a + 2;
        }
        System.out.println(y);
    }
}
```

22.4 Logical Operators – to Use, or not to Use: That is the Question!

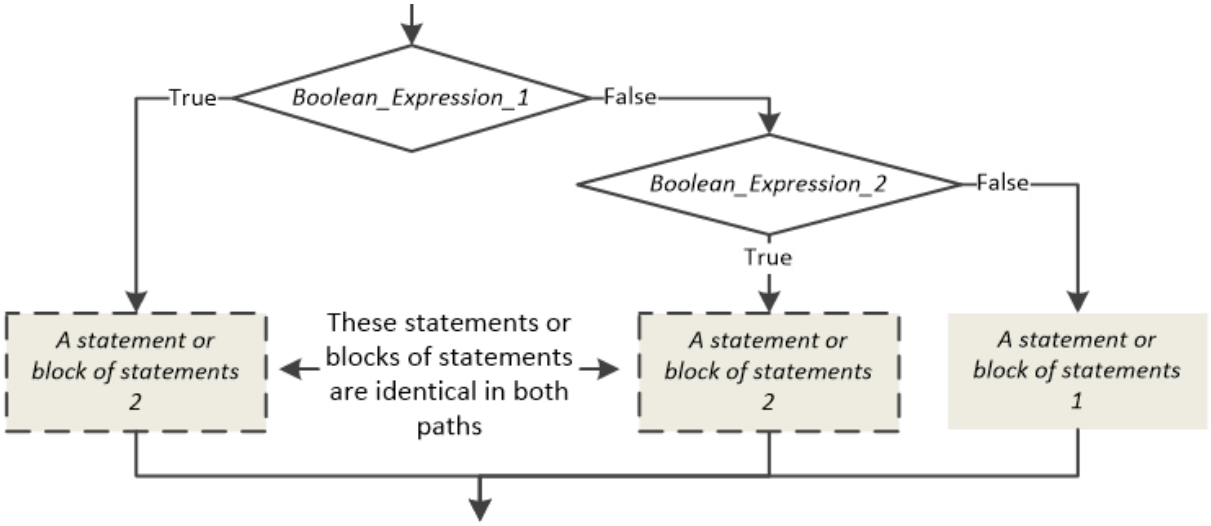
There are some cases in which you can use a logical operator instead of nested decision control structures, and this can lead to increased readability. Take a look at the following flowchart fragment given in general form.



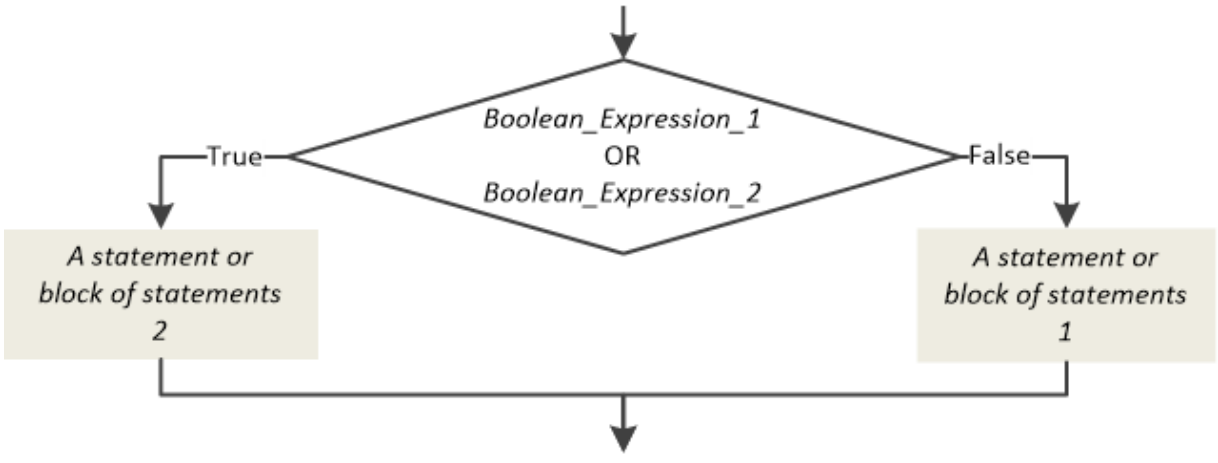
As you can see, the statement or block of statements 1 is executed only when **both** Boolean expressions evaluate to `true`. The statement or block of statements 2 is executed in all other cases. Therefore, this flowchart fragment can be redesigned using the `AND` logical operator.



Now, let's take a look at another flowchart fragment given in general form.



In this flowchart fragment, the statement or block of statements 2 is executed when **either** *Boolean_Expression_1* evaluates to `true` **or** *Boolean_Expression_2* evaluates to `true`. Therefore, you can redesign this flowchart fragment using the `OR` logical operator as shown here.



✎ *Obviously, these methodologies can be adapted to be used on nested decision control structures as well.*

Exercise 22.4-1 Rewriting the Code

Rewrite the following Java program using logical operators.

```
public static void main(String[] args) throws Exception {
   String today, name;
   today = cin.nextLine(); name = cin.nextLine();
   if (today.equals("February 16") == true) {
      if (name.equals("Loukia") == true) {
         System.out.println("Happy Birthday!!!");
      }
      else {
         System.out.println("No match!");
      }
   }
   else {
      System.out.println("No match!");
   }
}
```

Solution The `System.out.println("Happy Birthday!!!")` statement is executed only when both Boolean expressions evaluate to `true`. The statement `System.out.println("No match!")` is executed in all other cases. Therefore, you can rewrite the Java program using the AND (`&&`) logical operator.

```
public static void main(String[] args) throws Exception {
   String today, name;
   today = cin.nextLine(); name = cin.nextLine();
   if (today.equals("February 16") == true && name.equals("Loukia") == true) {
      System.out.println("Happy Birthday!!!");
   }
   else {
      System.out.println("No match!");
   }
}
```

Exercise 22.4-2 Rewriting the Code

Rewrite the following Java program using logical operators.

```
public static void main(String[] args) throws Exception {
   int a, b, y;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
```

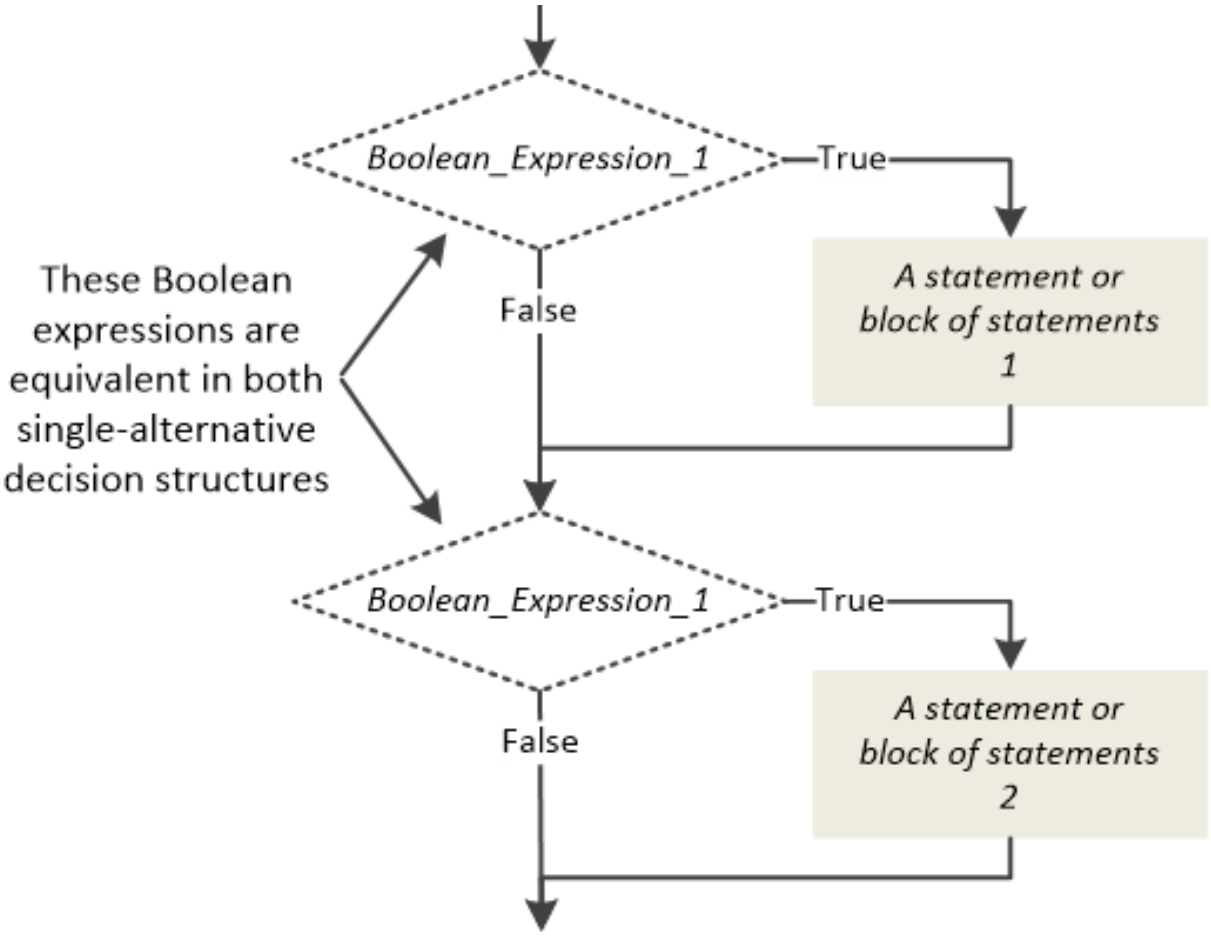
```
   y = 0;
   if (a > 10) {
      y++;
   }
   else if (b > 20) {
      y++;
   }
   else {
      y--;
   }
   System.out.println(y); }
```

Solution The y++ statement is executed when either variable a is greater than 10 or variable b is greater than 20. Therefore, you can rewrite the Java program using the OR (||) logical operator.

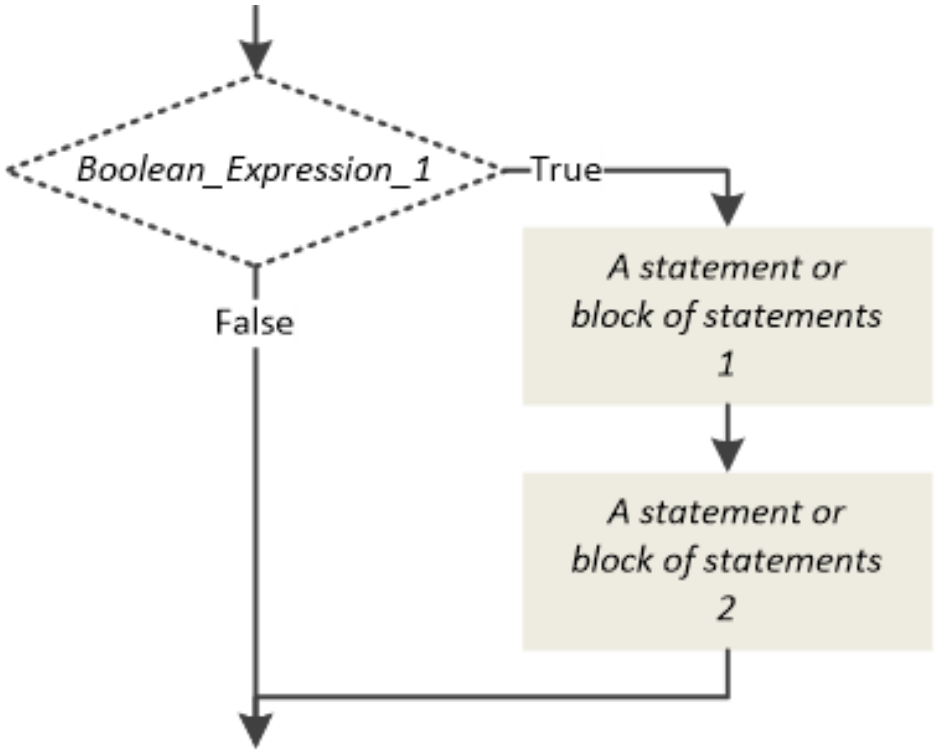
```
public static void main(String[] args) throws Exception {
   int a, b, y;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   y = 0;
   if (a > 10 || b > 20) {
      y++;
   }
   else {
      y--;
   }
   System.out.println(y); }
```

22.5 Merging Two or More Single-Alternative Decision Structures

Sometimes, you may design an algorithm that contains two or more single-alternative decision structures in a row, each of which evaluates the same Boolean expression. An example is shown here.



When a situation like this occurs, you can just merge all single-alternative decision structures to a single one, as follows.



The single-alternative decision structures need to be adjacent to each other. If any statement exists between them, you can't merge them unless you are able to move this statement to somewhere else in your code.

Exercise 22.5-1 Merging the Decision Control Structures

In the following Java program, merge the single-alternative decision structures.

```
public static void main(String[] args) throws Exception {
   int a;
   a = Integer.parseInt(cin.nextLine());
   if (a > 0) {
      System.out.println("Hello");
   }
   if (a > 0) {
      System.out.println("Hermes");
   }
}
```

Solution The first and second decision control structures are evaluating exactly the same Boolean expressions, so they can simply be merged into a single one.

The Java program becomes

```
public static void main(String[] args) throws Exception {
    int a;
    a = Integer.parseInt(cin.nextLine());
    if (a > 0) {
        System.out.println("Hello");
        System.out.println("Hermes");
    }
}
```

Exercise 22.5-2 Merging the Decision Control Structures

In the following Java program, merge as many single-alternative decision structures as possible.

```
public static void main(String[] args) throws Exception {
   int a, y, b;
   a = Integer.parseInt(cin.nextLine());
   y = 0;
   if (a > 0) {
      y += a + 1;
   }
   b = Integer.parseInt(cin.nextLine()); [More…]

   if (!(a <= 0)) {
      System.out.println("Hello Hera");
   }
   a++; [More…]

   if (a > 0) {
      System.out.println("Hallo Welt");
   }
   System.out.println(y); }
```

Solution Upon closer examination, it's evident that the first and second decision control structures are evaluating exactly the same Boolean expression. Specifically, negating `a > 0` gives `a <= 0`, and a second negation of `a <= 0` (using the NOT (`!`) operator this time) yields `!(a <= 0)`. Thus, `a > 0` is in fact equivalent to `!(a <= 0)`.

> *Two negations result in an affirmative.*

However, between the first and second decision control structures there is the statement `b = Integer.parseInt(cin.nextLine())`, which prevents you from merging them into a single one. Fortunately, this statement can be

moved to the beginning of the program since it doesn't really affect the rest of the flow of execution.

On the other hand, between the second and third decision control structures there is the statement a++, which also prevents you from merging; unfortunately, this statement cannot be moved anywhere else because it does affect the rest of the flow of execution (the second and third decision control structures are dependent upon this statement). Thus, the third decision control structure cannot be merged with the first and second ones!

The final Java program looks like this.

```
public static void main(String[] args) throws Exception {
   int a, b, y;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   y = 0;
   if (a > 0) {
      y += a + 1;
      System.out.println("Hello Hera");
   }
   a++;
   if (a > 0) {
      System.out.println("Hallo Welt");
   }
   System.out.println(y); }
```

22.6 Replacing Two Single-Alternative Decision Structures with a Dual-Alternative One

Take a look at the next example.

```
if (x > 40) {
   //Do something }
if (x <= 40) {
   //Do something else }
```

The first decision control structure evaluates variable x to test if it is bigger than 40, and right after that, a second decision control structure evaluates the same variable again to test if it is less than or equal to 40!

This is a very common “mistake” that novice programmers make. They use two single-alternative decision structures even though one dual-alternative decision structure can accomplish the same thing.

The previous example can be rewritten using only one dual-alternative decision structure, as shown here.

```
if (x > 40) {
```

```
   //Do something }
else {
   //Do something else }
```

Even though both examples are absolutely correct and work perfectly well, the second alternative is better. The CPU needs to evaluate only one Boolean expression, which results in faster execution time.

✎ *The two single-alternative decision structures must be adjacent to each other. If any statement exists between them, you can't "merge" them (that is, replace them with a dual-alternative decision structure) unless you can move this statement to somewhere else in your code.*

Exercise 22.6-1 "Merging" the Decision Control Structures

In the following Java program, "merge" as many single-alternative decision structures as possible.

```
public static void main(String[] args) throws Exception {
   int a, y, b;
   a = Integer.parseInt(cin.nextLine());
   y = 0;
   if (a > 0) {
      y += a;
   }
   b = Integer.parseInt(cin.nextLine());
   if (!(a > 0)) {
      System.out.println("Hello Zeus");
   }
   if (y > 0) {
      System.out.println(y + 5);
   }
   y += a;
   if (y <= 0) {
      System.out.println(y + 12);
   }
}
```

***Solution** The first decision control structure evaluates variable `a` to test if it is greater than zero, and just right after that the second decision control structure evaluates variable `a` again to test if it is not greater than zero. Even though there is the statement `b = Integer.parseInt(cin.nextLine())` between them, this statement can be moved somewhere else because it doesn't really affect the rest of the flow of execution. Therefore, the first and second decision control structures can be merged!*

On the other hand, between the third and fourth decision control structures there is the statement `y += a` which prevents you from merging. This statement cannot be moved anywhere else because it does affect the rest of the flow of execution (the third and fourth decision control structures are dependent upon this statement). Therefore, the third and fourth decision control structures cannot be merged!

The final Java program becomes

```
public static void main(String[] args) throws Exception {
    int a, b, y;
    a = Integer.parseInt(cin.nextLine()); b =
    Integer.parseInt(cin.nextLine());
    y = 0;
    if (a > 0) {
        y += a;
    }
    else {
        System.out.println("Hello Zeus");
    }
    if (y > 0) {
        System.out.println(y + 5);
    }
    y += a;
    if (y <= 0) {
        System.out.println(y + 12);
    }
}
```

22.7 Put the Boolean Expressions Most Likely to be True First

Both the multiple-alternative and the case decision structure often need to check several Boolean expressions before deciding which statement or block of statements to execute. In the next decision control structure,

```
if (Boolean_Expression_1) {
    A statement or block of statements 1
}
else if (Boolean_Expression_2) {
    A statement or block of statements 2
```

}
else if (*Boolean_Expression_3*) {

```
    A statement or block of statements 3
```

}

the program first tests if *Boolean_Expression_1* is `true`. If not, it tests if *Boolean_Expression_2* is `true`, and if not, it tests *Boolean_Expression_3*. However, what if *Boolean_Expression_1* is `false` most of the time and *Boolean_Expression_3* is `true` most of the time? This means that time is wasted testing *Boolean_Expression_1*, which is usually `false`, before testing *Boolean_Expression_3*, which is usually `true`.

To make your programs more efficient, you can put the Boolean expressions that are most likely to be `true` at the beginning, and the Boolean expressions that are most likely to be `false` at the end, as follows.

```
if (Boolean_Expression_3) {
    A statement or block of statements 3
}
else if (Boolean_Expression_2) {
    A statement or block of statements 2
}
else if (Boolean_Expression_1) {
    A statement or block of statements 1
}
```

Although this change may seem nonessential, every little bit of time that you save can add up to make your programs run faster and more efficiently.

Exercise 22.7-1 Rearranging the Boolean Expressions

According to research, America's favorite pets are dogs, with cats at second place, guinea pigs next, and parrots coming in last. In the following Java program, rearrange the Boolean expressions to make the program run faster and more efficiently for most of the cases.

```
public static void main(String[] args) throws Exception {
   String kind;
   System.out.print("What is your favorite pet? "); kind = cin.nextLine();
   switch (kind) {
      case "Parrots":
```

```
            System.out.println("It screeches!");
            break;
         case "Guinea pig":
            System.out.println("It squeaks");
            break;
         case "Dog":
            System.out.println("It barks");
            break;
         case "Cat":
            System.out.println("It meows");
            break;
      }
}
```

Solution For this research, you can rearrange the Java program to make it run a little bit faster for most of the cases.

```
public static void main(String[] args) throws Exception {
   String kind;
   System.out.print("What is your favorite pet? "); kind = cin.nextLine();
   switch (kind) {
      case "Dog":
         System.out.println("It barks");
         break;
      case "Cat":
         System.out.println("It meows");
         break;
      case "Guinea pig":
         System.out.println("It squeaks");
         break;
      case "Parrots":
         System.out.println("It screeches!");
         break;
   }
}
```

22.8 Why is Code Indentation so Important?

As you've been reading through this book, you may wonder why space characters appear in front of the Java statements and why these statements are not written at the leftmost edge of the paragraph, as in the following example.

```
public static void main(String[] args) throws Exception {
int x, y;
System.out.print("Enter a number: "); x = Integer.parseInt(cin.nextLine());
System.out.print("Enter a second number: "); y = Integer.parseInt(cin.nextLine()); if (x
```

```
> 5) {
System.out.println("Variable x is greater than 5"); }
else {
x = x + y;
System.out.println("Hello Zeus!"); if (x == 3) {
System.out.println("Variable x contains a value of 3"); }
else {
x = x - y;
System.out.println("Hello Olympians!"); if (x + y == 24) {
System.out.println("The sum of x + y is equal to 24"); }
else {
System.out.println("Nothing of the above"); }
}
}
}
```

The answer is obvious! A code without indentation is difficult to read and understand. Anyone who reads a code written this way gets confused about the `if – else` pairing (that is, to which `if` an `else` belongs). Moreover, if a long Java program is written this way, it is almost impossible to find, for example, the location of a forgotten closing brace }.

Code indentation can be defined as a way to organize your source code. Indentation formats the code using spaces or tabs in order to improve readability. Well indented code is very helpful, even if it takes some extra effort, because in the long run it saves you a lot of time when you revisit your code. Unfortunately, it is sometimes overlooked and the trouble occurs at a later time. Following a particular programming style helps you to avoid syntax and logic errors. It also helps programmers to more easily study and understand code written by others.

> ✎ *Code indentation is similar to the way authors visually arrange the text of a book. Instead of writing long series of sentences, they break the text into chapters and paragraphs. This action doesn't change the meaning of the text but it makes it easier to read.*

All statements that appear inside a set of braces { } should always be indented. For example, by indenting the statements inside a dual-alternative decision structure, you visually set them apart. As a result, anyone can tell at a glance which statements are executed when the Boolean expression evaluates to `true`, and which are executed when the Boolean expression evaluates to `false`.

✎ *Only humans have difficulty reading and understanding a program without indentation. A computer can execute any Java code, written with or without indentation, as long as it contains no syntax errors.*

22.9 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Smaller algorithms are always the best solution to a given problem.
2) You can always move a statement outside, and right before, a dual-alternative decision structure as long as it exists at the beginning of both paths of the decision structure.
3) You can always use a logical operator instead of nested decision control structures to increase readability.
4) Two single-alternative decision structures can be merged into one single-alternative decision only when they are in a row and when they evaluate equivalent Boolean expressions.
5) Conversion from a dual-alternative decision structure to two single-alternative decision structures is always possible.
6) Two single-alternative decision structures can be replaced by one dual-alternative decision only when they are in a row and only when they evaluate the same Boolean expression.
7) Java programs that include decision control structures and written without code indentation cannot be executed by a computer.

22.10 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The following two programs

```
public static void main(String[] args) throws Exception {
    int a;
    a = Integer.parseInt(cin.nextLine()); if (a > 40) {
        System.out.println(a * 2);
        a++;
    }
    else {
        System.out.println(a * 2);
        a += 5;
```

```
    }
}
```

```
public static void main(String[] args) throws Exception {
   int a;
   a = Integer.parseInt(cin.nextLine()); System.out.println(a * 2); if (a > 40) {
      a++;
   }
   else {
      a += 5;
   }
}
```

a) produce the same result.

b) do not produce the same result.

c) none of the above

2) The following two programs

```
public static void main(String[] args) throws Exception {
    int a;
    a = Integer.parseInt(cin.nextLine()); if (a > 40) {
        System.out.println(a * 2);
    }
    if (a > 40) {
        System.out.println(a * 3);
    }
}
```

```
public static void main(String[] args) throws Exception {
   int a;
   a = Integer.parseInt(cin.nextLine()); if (a > 40) {
      System.out.println(a * 2);
      System.out.println(a * 3);
   }
}
```

a) produce the same results, but the first program is faster.

b) produce the same results, but the second program is faster.

c) do not produce the same results.

d) none of the above

3) The following two programs

```
public static void main(String[] args) throws Exception {
    int a;
```

```
        a = Integer.parseInt(cin.nextLine()); if (a > 40) {
            System.out.println(a * 2);
        }
        else {
            System.out.println(a * 3);
        }
    }
```

```
public static void main(String[] args) throws Exception {
   int a;
   a = Integer.parseInt(cin.nextLine()); if (a > 40) {
      System.out.println(a * 2);
   }
   if (a <= 40) {
      System.out.println(a * 3);
   }
}
```

a) produce the same result(s), but the first program is faster.

b) produce the same result(s), but the second program is faster.

c) do not produce the same result(s).

d) none of the above

4) The following program

public static void main(**String**[] args) throws Exception {
int x; x = Integer.**parseInt**(cin.**nextLine**()); **if** (x < 0) x = (-1) * x;
System.out.**println**(x); }

cannot be executed by a computer because a) it does not use code indentation.

b) it includes logic errors.

c) none of the above

22.11 Review Exercises

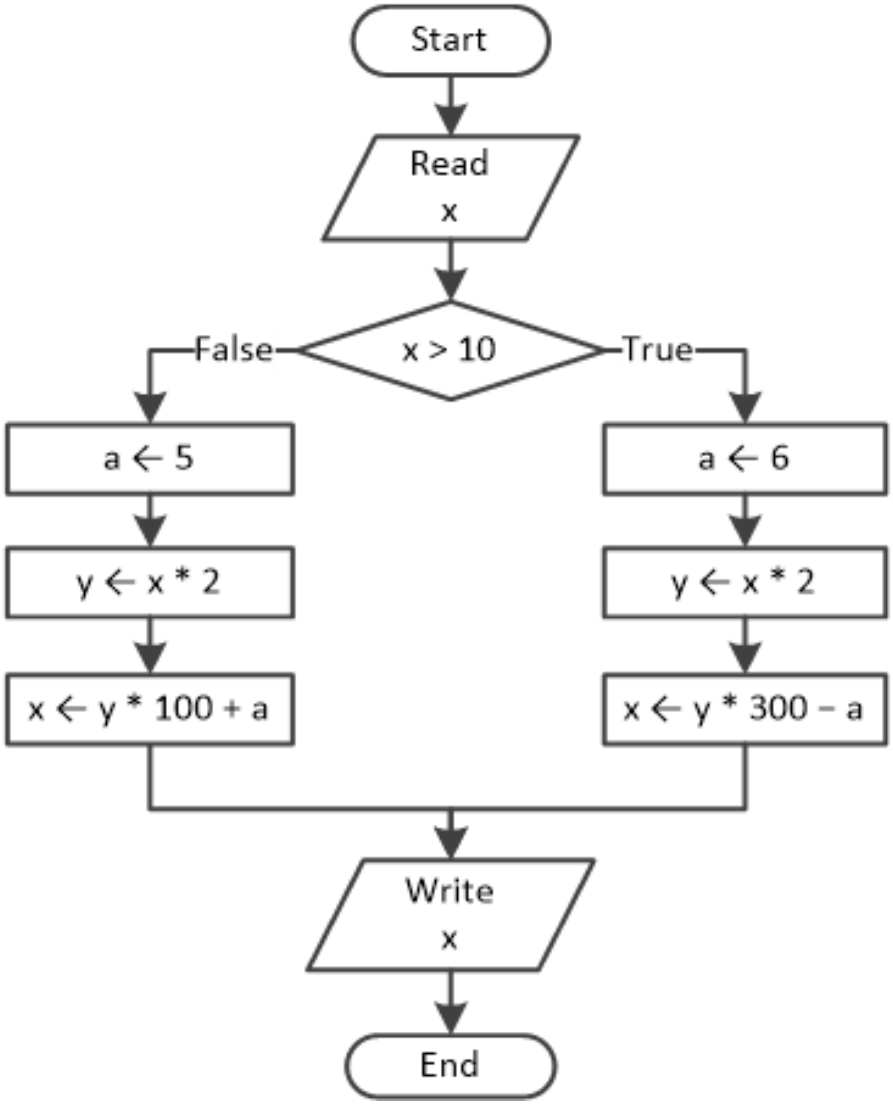
Complete the following exercises.

1) Rewrite the following Java program using fewer statements.

```
public static void main(String[] args) throws Exception {
   int a, x, y;
   y = Integer.parseInt(cin.nextLine());
   if (y > 0) {
      x = Integer.parseInt(cin.nextLine());
      a = x * 4 * y;
```

```
      System.out.println(y);
      a++;
   }
   else {
      x = Integer.parseInt(cin.nextLine());
      a = x * 2 * y + 7;
      System.out.println(y);
      a--;
   }
   System.out.println(a); }
```

2) Redesign the following flowchart using fewer statements.



3) Rewrite the following Java program using fewer statements.

```
public static void main(String[] args) throws Exception {
   double a, y;
   a = Double.parseDouble(cin.nextLine());
```

```
   if (a < 1) {
      y = 5 + a;
      System.out.println(y);
   }
   else if (a < 5) {
      y = 23 / a;
      System.out.println(y);
   }
   else if (a < 10) {
      y = 5 * a;
      System.out.println(y);
   }
   else {
      System.out.println("Error!");
   }
}
```

4) Rewrite the following Java program using logical operators.

```
public static void main(String[] args) throws Exception {
   int day, month; String name;
   day = Integer.parseInt(cin.nextLine()); month =
   Integer.parseInt(cin.nextLine()); name = cin.nextLine();
   if (day == 16) {
      if (month == 2) {
         if (name.equals("Loukia") == true) {
            System.out.println("Happy Birthday!!!");
         }
         else {
            System.out.println("No match!");
         }
      }
      else {
         System.out.println("No match!");
      }
   }
   else {
      System.out.println("No match!");
   }
}
```

5) A teacher asks her students to rewrite the following Java program without using logical operators.

```
public static void main(String[] args) throws Exception {
   double a, b, c, d;
   a = Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine());
   c = Double.parseDouble(cin.nextLine());
```

```
   if (a > 10 && c < 2000) {
      d = (a + b + c) / 12;
      System.out.println("The result is: " + d);
   }
   else {
      System.out.println("Error!");
   }
}
```

One student wrote the following Java program:

```
public static void main(String[] args) throws Exception {
    double a, b, c, d;
    a = Double.parseDouble(cin.nextLine()); b =
    Double.parseDouble(cin.nextLine()); c =
    Double.parseDouble(cin.nextLine());
    if (a > 10) {
        if (c < 2000) {
            d = (a + b + c) / 12;
            System.out.println("The result is: " + d);
        }
        else {
            System.out.println("Error!");
        }
    }
}
```

Determine if the program operates the same way for all possible paths as the one provided by the teacher. If not, try to modify it and make it work the same way.

6) Rewrite the following Java program using only single-alternative decision structures.

```
public static void main(String[] args) throws Exception {
   double a, b, c, d;
   a = Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine());
   c = Double.parseDouble(cin.nextLine());
   if (a > 10) {
      if (b < 2000) {
         if (c != 10) {
            d = (a + b + c) / 12;
            System.out.println("The result is: " + d);
         }
      }
```

```
   }
   else {
      System.out.println("Error!");
   }
}
```

7) In the following Java program, replace the two single-alternative decision structures by one dual-alternative decision structure.

```
public static void main(String[] args) throws Exception {
   int a, b, y;
   a = Integer.parseInt(cin.nextLine());
   y = 3;
   if (a > 0) {
      y = y * a;
   }
   b = Integer.parseInt(cin.nextLine()); if (!(a <= 0)) {
      System.out.println("Hello Zeus");
   }
   System.out.println(y + " " + b); }
```

8) Rewrite the following Java program, using only one dual-alternative decision structure.

```
public static void main(String[] args) throws Exception {
   double a, b, y;
   a = Double.parseDouble(cin.nextLine());
   y = 0;
   if (a > 0) {
      y = y + 7;
   }
   b = Double.parseDouble(cin.nextLine()); if (!(a > 0)) {
      System.out.println("Hello Zeus");
   }
   if (a <= 0) {
      System.out.println(Math.abs(a));
   }
   System.out.println(y); }
```

9) According to research from 2013, the most popular operating system on tablet computers was iOS, with Android being in second place and Microsoft Windows in last place. In the following Java program, rearrange the Boolean expressions to make the program run more efficiently for most of the cases.

```
public static void main(String[] args) throws Exception {
   String os;
   System.out.print("What is your tablet's OS? "); os = cin.nextLine();
```

```
   if (os.equals("Windows") == true) {
      System.out.println("Microsoft");
   }
   else if (os.equals("iOS") == true) {
      System.out.println("Apple");
   }
   else if (os.equals("Android") == true) {
      System.out.println("Google");
   }
}
```

Chapter 23
More with Decision Control Structures

23.1 Simple Exercises with Decision Control Structures

Exercise 23.1-1 Is it an Integer?

Write a Java program that prompts the user to enter a number and then displays a message indicating whether the data type of this number is integer or real.

Solution

It is well known that a number is considered an integer when it contains no fractional part. In Java, you can use the `(int)` casting operator to get the integer portion of any real number. If the user-provided number is equal to its integer portion, then the number is considered an integer.

For example, if the user enters the number 7, this number and its integer portion, `(int)(7)`, are equal.

On the other hand, if the user enters the number 7.3, this number and its integer portion, `(int)(7.3)`, are not equal.

The Java program is as follows.

Project_23.1-1

```
public static void main(String[] args) throws Exception {
   double x;
   System.out.print("Enter a number: "); x = Double.parseDouble(cin.nextLine());
   if (x == (int)x) {
      System.out.println(x + " is integer");
   }
   else {
      System.out.println(x + " is real");
   }
}
```

✎ *Variable* `x` *is declared as* `double` *and the method* `Double.parseDouble()` *is used in the data input stage. This is necessary in order to allow the user to enter either an integer or a float.*

Exercise 23.1-2 Validating Data Input and Finding Odd and Even Numbers

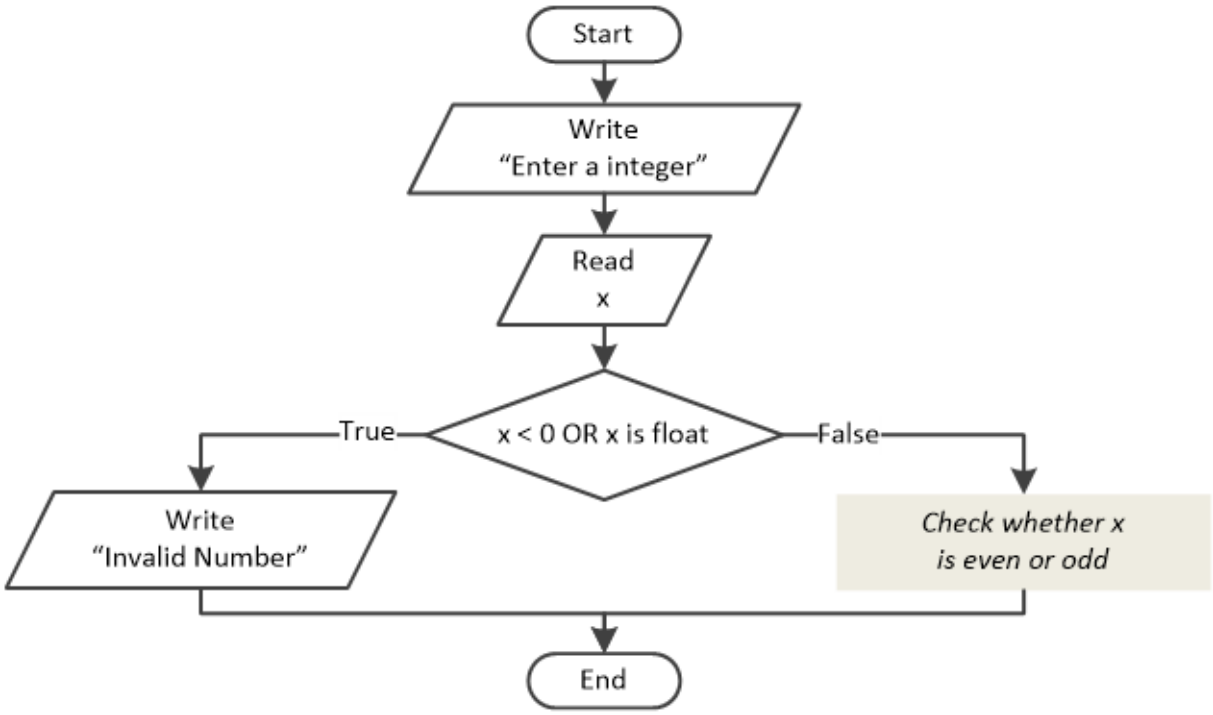
Design a flowchart and write the corresponding Java program that prompts the user to enter a non-negative integer, and then displays a message indicating whether this number is even; it must display "Odd" otherwise. Moreover, if the user enters a negative value or a float, an error message must be displayed.

(This exercise gives you some practice in working with data validation).

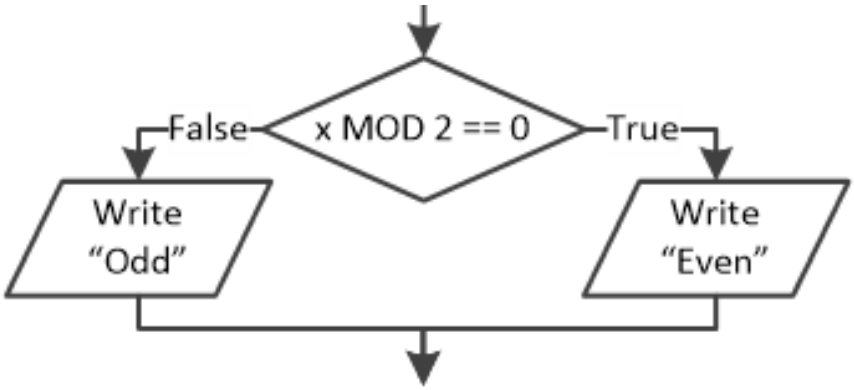
Solution

Data validation is the process of restricting data input, forcing the user to enter only valid values.

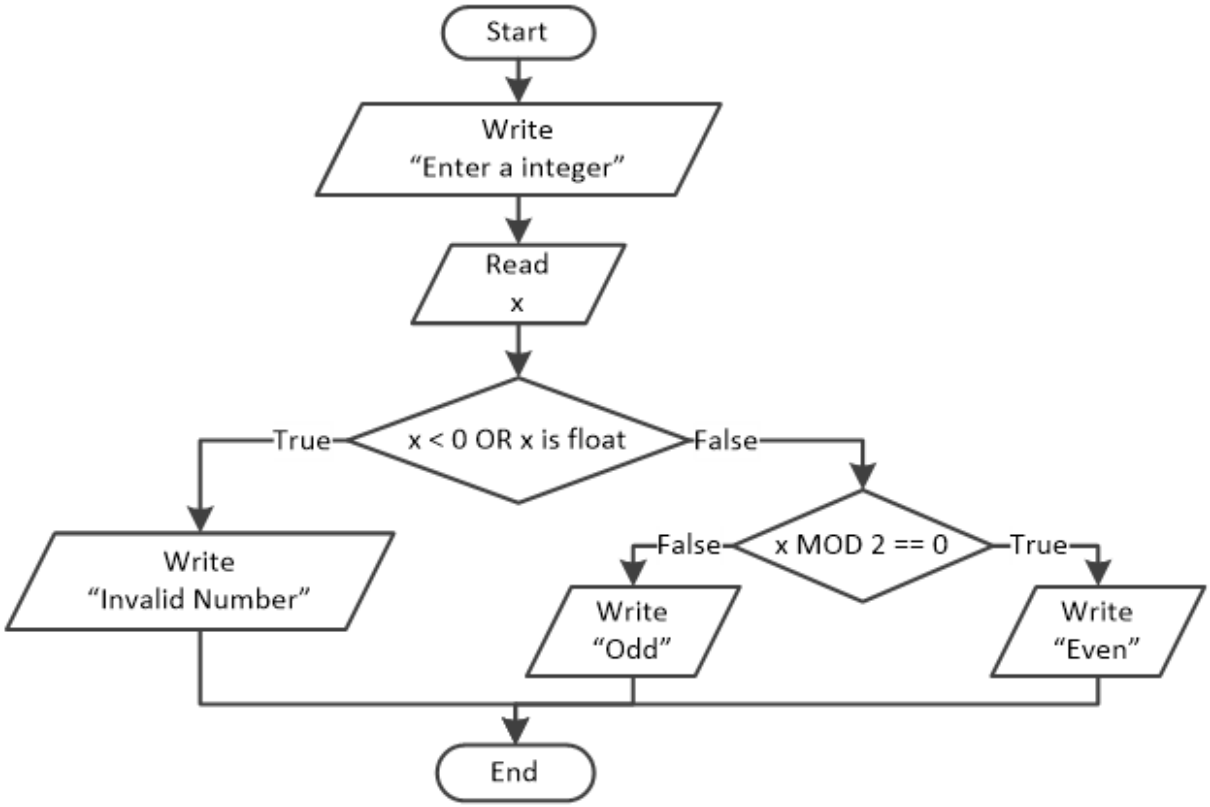
In this exercise, you need to prompt the user to enter a non-negative integer and display an error message when they enter either a negative value or a float. The flowchart that solves this exercise given in general form is as follows.



The following decision control structure is taken from Exercise 17.1-4. It tests whether variable x is even or odd.



After combining both flowcharts, the final flowchart looks like this.



The Java program is shown here.

```
Project_23.1-2a
public static void main(String[] args) throws Exception {
                        double x;
        System.out.print("Enter an integer: "); x =
            Double.parseDouble(cin.nextLine());
                if (x < 0 || x != (int)x) {
             System.out.println("Invalid Number");
                        }
```

```
                else {
                    if (x % 2 == 0) {
                    System.out.println("Even");
                    }
                    else {
                    System.out.println("Odd");
                    }
                }
            }
```

Instead of using nested decision structures, you can alternatively use a multiple-alternative decision structure, as shown here.

```
                Project_23.1-2b
public static void main(String[] args) throws Exception {
                    double x;
        System.out.print("Enter an integer: "); x =
            Double.parseDouble(cin.nextLine());
                if (x < 0 || x != (int)x) {
                System.out.println("Invalid Number");
                    }
                else if (x % 2 == 0) {
                System.out.println("Even");
                    }
                else {
                System.out.println("Odd");
                    }
                }
```

Exercise 23.1-3 Where is the Tollkeeper?

In a toll gate, there is an automatic system that recognizes whether the passing vehicle is a motorcycle, a car, or a truck. Write a Java program that lets the user enter the type of the vehicle (M for motorcycle, C for car, and T for truck) and then displays the corresponding amount of money the driver must pay according to the following table.

Vehicle Type	Amount to Pay
Motorcycle	$1

Car	$2
Track	$4

The program must function properly even when characters are entered in lowercase. For example, the program must function properly either for "M" or "m". However, if the user enters a character other than M, C, or T (uppercase or lowercase), an error message must be displayed.

(Some more practice with data validation!) *Solution*

The solution to this problem is quite simple. The only thing that needs attention is that the user may enter the uppercase letters M, C, or T, or the lowercase letters m, c, or t. The program needs to accept both. To handle this, you can convert the user's input to uppercase using the `toUpperCase()` method. Then you need to check only for the M, C, or T characters in uppercase.

The Java program is shown here.

Project_23.1-3a

```
public static void main(String[] args) throws Exception {
                        String v;
            v = cin.nextLine().toUpperCase();
     //You need to check only for capital M, C, and T
         if (!v.equals("M") && !v.equals("C") &&
                    !v.equals("T")) {
           System.out.println("Invalid vehicle");
                        }
              else if (v.equals("M")) {
          System.out.println("You need to pay $1");
                        }
              else if (v.equals("C")) {
          System.out.println("You need to pay $2");
                        }
              else if (v.equals("T")) {
          System.out.println("You need to pay $4");
                        }
                      }
```

✎ *Note how Java converts the user's input to uppercase.*

✎ *The expression* `!v.equals("M")` *is equivalent to the expression* `v.equals("M") == false` *and the expression* `v.equals("M")` *is equivalent to the expression* `v.equals("M") == true`.

However, this exercise can be solved slightly more efficiently, if you move the first case of the multiple-alternative decision structure to the end, as shown here.

```
                    Project_23.1-3b
public static void main(String[] args) throws Exception {
                          String v;
              v = cin.nextLine().toUpperCase();
                    if (v.equals("M")) {
            System.out.println("You need to pay $1");
                          }
                else if (v.equals("C")) {
            System.out.println("You need to pay $2");
                          }
                else if (v.equals("T")) {
            System.out.println("You need to pay $4");
                          }
                      else {
             System.out.println("Invalid vehicle");
                          }
                        }
```

Exercise 23.1-4 The Most Scientific Calculator Ever!

*Write a Java program that emulates the way an electronic calculator functions. The program must first prompt the user to enter a number, then the type of operation (+, −, *, /), and finally a second number. Subsequently, the program must perform the chosen operation and display the result. However, if the user enters an operand other than +, −, *, or /, an error message must be displayed.*

Solution

The only thing that you need to take care of in this exercise is the possibility the user could enter zero for the divisor (the second number). As you know from mathematics, division by zero is not possible.

The following Java program uses the case decision structure to check the type of operation.

Project_23.1-4

```
public static void main(String[] args) throws Exception {
   double a, b; String op;
   System.out.print("Enter 1st number: "); a = Double.parseDouble(cin.nextLine());
   System.out.print("Enter type of operation: "); op = cin.nextLine(); //Variable op is
   of type String System.out.print("Enter 2nd number: "); b =
   Double.parseDouble(cin.nextLine());
   switch (op) {
      case "+":
         System.out.println(a + b);
         break;
      case "-":
         System.out.println(a - b);
         break;
      case "*":
         System.out.println(a * b);
         break;
      case "/":
         if (b == 0) {
            System.out.println("Error: Division by zero");
         }
         else {
            System.out.println(a / b);
         }
         break;
      default:
         System.out.println("Error: Invalid operand");
   }
}
```

Exercise 23.1-5 Converting Gallons to Liters, and Vice Versa

Write a Java program that displays the following menu: 1) Convert gallons to liters 2) Convert liters to gallons The program must then prompt the user to enter a choice (of 1 or 2) and a quantity, and subsequently calculate and display the required value. It is given that 1 gallon = 3.785 liters Solution

The Java program is shown here.

Project_23.1-5

```
static final double COEFFICIENT = 3.785;
public static void main(String[] args) throws Exception {
   int choice; double quantity, result;
```

```
    System.out.println("1: Gallons to liters"); System.out.println("2: Liters to
    gallons"); System.out.print("Enter choice: "); choice =
    Integer.parseInt(cin.nextLine());
    System.out.print("Enter quantity: "); quantity = Double.parseDouble(cin.nextLine());
    if (choice == 1) {
       result = quantity * COEFFICIENT;
       System.out.println(quantity + " gallons = " + result + " liters");
    }
    else {
       result = quantity / COEFFICIENT;
       System.out.println(quantity + " liters = " + result + " gallons");
    }
}
```

Exercise 23.1-6 Converting Gallons to Liters, and Vice Versa (with Data Validation)

Rewrite the Java program of the previous exercise to validate the data input. A different error message for each type of input error must be displayed when the user enters a choice other than 1 or 2, or a negative gas quantity.

Solution

The following Java program, given in general form, solves this exercise. It prompts the user to enter a choice. If the choice is invalid, it displays an error message; otherwise, it prompts the user to enter a quantity. However, if the quantity entered is invalid too, it displays another error message; otherwise it proceeds to data conversion, depending on the user's choice.

```
Main Code
        static final double COEFFICIENT = 3.785;
public static void main(String[] args) throws Exception {
            int choice; double quantity, result;
          System.out.println("1: Gallons to liters");
          System.out.println("2: Liters to gallons");
         System.out.print("Enter choice: "); choice =
              Integer.parseInt(cin.nextLine());
              if (choice != 1 && choice != 2) {
              System.out.println("Wrong choice!");
                              }
                            else {
             System.out.print("Enter quantity: ");
          quantity = Double.parseDouble(cin.nextLine());
```

```
if (quantity < 0) {
   System.out.println("Invalid quantity!");
}
else {
   Code Fragment 1: Convert gallons to liters or
   liters to gallons depending on user's choice.
}
}
}
```

Code Fragment 1 shown below is taken from the previous exercise (Exercise 23.1-5). It converts gallons to liters, or liters to gallons, depending on the user's choice.

Code Fragment 1

```
if (choice == 1) {
   result = quantity * COEFFICIENT; System.out.println(quantity + " gallons = " + result
   + " liters"); }
else {
   result = quantity / COEFFICIENT; System.out.println(quantity + " liters = " + result
   + " gallons"); }
```

After embedding **Code Fragment 1** in **Main Code**, the final Java program becomes Project_23.1-6

```
static final double COEFFICIENT = 3.785;
public static void main(String[] args) throws Exception {
   int choice; double quantity, result;
   System.out.println("1: Gallons to liters"); System.out.println("2: Liters to
   gallons"); System.out.print("Enter choice: "); choice =
   Integer.parseInt(cin.nextLine());
   if (choice != 1 && choice != 2) {
      System.out.println("Wrong choice!");
   }
   else {
      System.out.print("Enter quantity: ");
      quantity = Double.parseDouble(cin.nextLine());
      if (quantity < 0) {
         System.out.println("Invalid quantity!");
      }
      else {
```

```
        if (choice == 1) { [More…]
           result = quantity * COEFFICIENT;
           System.out.println(quantity + " gallons = " + result + " liters");
        }
        else {
           result = quantity / COEFFICIENT;
           System.out.println(quantity + " liters = " + result + " gallons");
        }
      }
   }
}
```

23.2 Finding Minimum and Maximum Values with Decision Control Structures

Suppose there are some men and you want to find the lightest one. Let's say that each one of them comes by and tells you his weight. What you must do is, memorize the weight of the first man that has come by and for each new man, you have to compare his weight with the one that you keep memorized. If he is heavier, you ignore his weight. However, if he is lighter, you need to forget the previous weight and memorize the new one. The same procedure continues until all the men have come by.

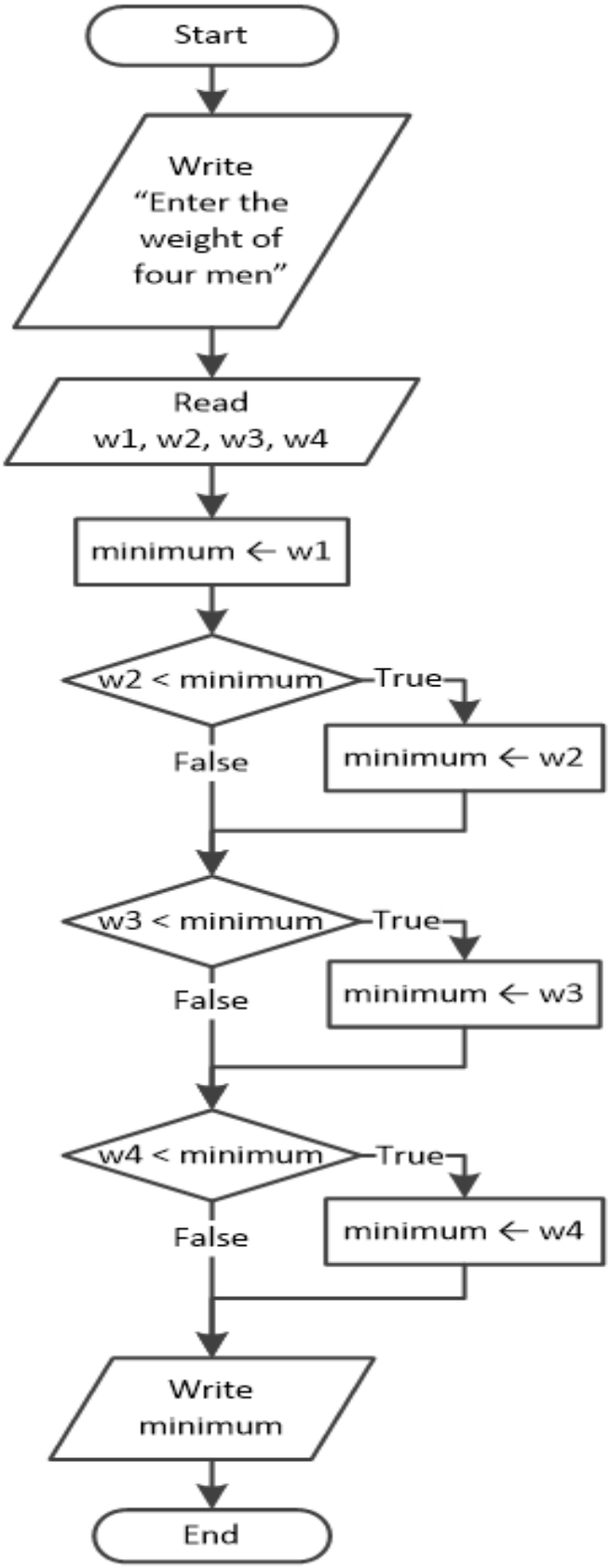
Let's ask four men to come by in a random order. Assume that their weights, in order of appearance, are 165, 170, 160, and 180 pounds.

Procedure	Value of Variable `minimum` in Your Mind!
The first man comes by. He weighs 165 pounds. Keep his weight in your mind (imagine a variable in your mind named `minimum`).	`minimum = 165`
The second man comes by. He weighs 170 pounds. He does not weigh less than the weight you are keeping in variable `minimum`, so you must ignore his weight. Variable `minimum` in your mind still contains the value 165.	`minimum = 165`
The third man comes by. He weighs 160 pounds, which is less than the weight you are keeping in variable `minimum`, so you	`minimum = 160`

must forget the previous value and keep the value 160 in variable `minimum`.	
The fourth man comes by. He weighs 180 pounds. He does not weigh less than the weight you are keeping in variable `minimum`, so you must ignore his weight. Variable `minimum` still contains the value 160.	`minimum = 160`

When the procedure finishes, the variable `minimum` in your mind contains the weight of the lightest man!

Following are the flowchart and the corresponding Java program that prompts the user to enter the weight of four men and then finds and displays the lightest weight.


Start
Write "Enter the weight of four men"
Read w1, w2, w3, w4
minimum ← w1
w2 < minimum
True
minimum ← w2
False
w3 < minimum
True
minimum ← w3
False
w4 < minimum
True
minimum ← w4
False
Write minimum
End

Project_23.2

```
public static void main(String[] args) throws Exception {
   int w1, w2, w3, w4, minimum; System.out.print("Enter the weight ");
   System.out.println("of four men:");
   w1 = Integer.parseInt(cin.nextLine()); w2 = Integer.parseInt(cin.nextLine()); w3 =
   Integer.parseInt(cin.nextLine()); w4 = Integer.parseInt(cin.nextLine());
   //Memorize the weight of the first man minimum = w1;
   //If second man is lighter, forget previous //value and memorize his weight if (w2 <
   minimum) {
      minimum = w2;
   }
   //If third man is lighter, forget previous //value and memorize his weight if (w3 <
   minimum) {
      minimum = w3;
   }
   //If fourth man is lighter, forget previous //value and memorize his weight if (w4 <
   minimum) {
      minimum = w4;
   }
   System.out.println(minimum); }
```

✎ *Note that this program is trying to find out the lowest value and* ***not*** *which variable this value was actually assigned to.*

✎ *You can find the maximum instead of the minimum value by simply replacing the "less than" with a "greater than" operator in all Boolean expressions.*

Exercise 23.2-1 Finding the Name of the Heaviest Person

Write a Java program that prompts the user to enter the weights and the names of three people and then displays the name and the weight of the heaviest person.

Solution

In this exercise, along with the maximum weight, you need to store in another variable the name of the person who actually has that weight. The Java program is shown here.

Project_23.2-1

```
public static void main(String[] args) throws Exception {
   int w1, w2, w3, maximum; String n1, n2, n3, mName;
   System.out.print("Enter the weight of the 1st person: "); w1 =
   Integer.parseInt(cin.nextLine());
```

```
System.out.print("Enter the name of the 1st person: "); n1 = cin.nextLine();
System.out.print("Enter the weight of the 2nd person: "); w2 =
Integer.parseInt(cin.nextLine());
System.out.print("Enter the name of the 2nd person: "); n2 = cin.nextLine();
System.out.print("Enter the weight of the 3rd person: "); w3 =
Integer.parseInt(cin.nextLine());
System.out.print("Enter the name of the 3rd person: "); n3 = cin.nextLine();
maximum = w1; //Memorize the weight mName = n1; //and the name of the first person.
if (w2 > maximum) { //If second person is heavier, forget previous values, and
maximum = w2; //memorize the weight
   mName = n2; //and the name of the second person.
}
if (w3 > maximum) { //If third person is heavier, forget previous values, and
maximum = w3; //memorize the weight
   mName = n3; //and the name of the third person.
}
System.out.println("The heaviest person is " + mName); System.out.println("Their
weight is " + maximum); }
```

✎ *In case the two heaviest people happen to have the same weight, the name of the first one in order is found and displayed.*

23.3 Decision Control Structures in Solving Mathematical Problems

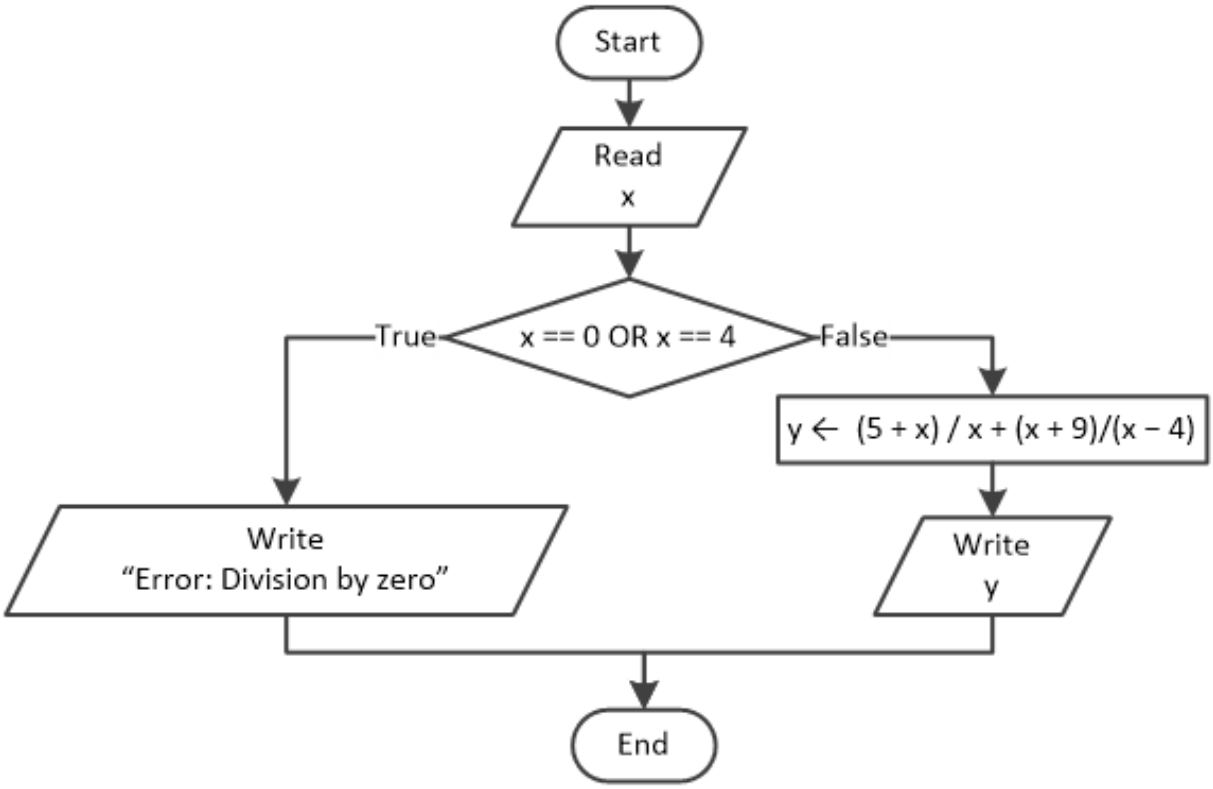
Exercise 23.3-1 Finding the Value of y

Design a flowchart and write the corresponding Java program that finds and displays the value of y (if possible) in the following formula.

$$y = \frac{5 + x}{x} + \frac{x + 9}{x - 4}$$

Solution

In this exercise, it's crucial to prevent the user from entering values of 0 or 4, as they result in a zero denominator. Therefore, the program needs to take these restrictions into consideration. The flowchart is shown here.



and the Java program is shown here.

Project_23.3-1

```
public static void main(String[] args) throws Exception {
   double x, y;
   x = Double.parseDouble(cin.nextLine());
   if (x == 0 || x == 4) {
      System.out.println("Error: Division by zero!");
   }
   else {
      y = (5 + x) / x + (x + 9) / (x - 4);
      System.out.println(y);
   }
}
```

Exercise 23.3-2 Finding the Values of y

Design a flowchart and write the corresponding Java program that finds and displays the values of y (if possible) in the following formula.

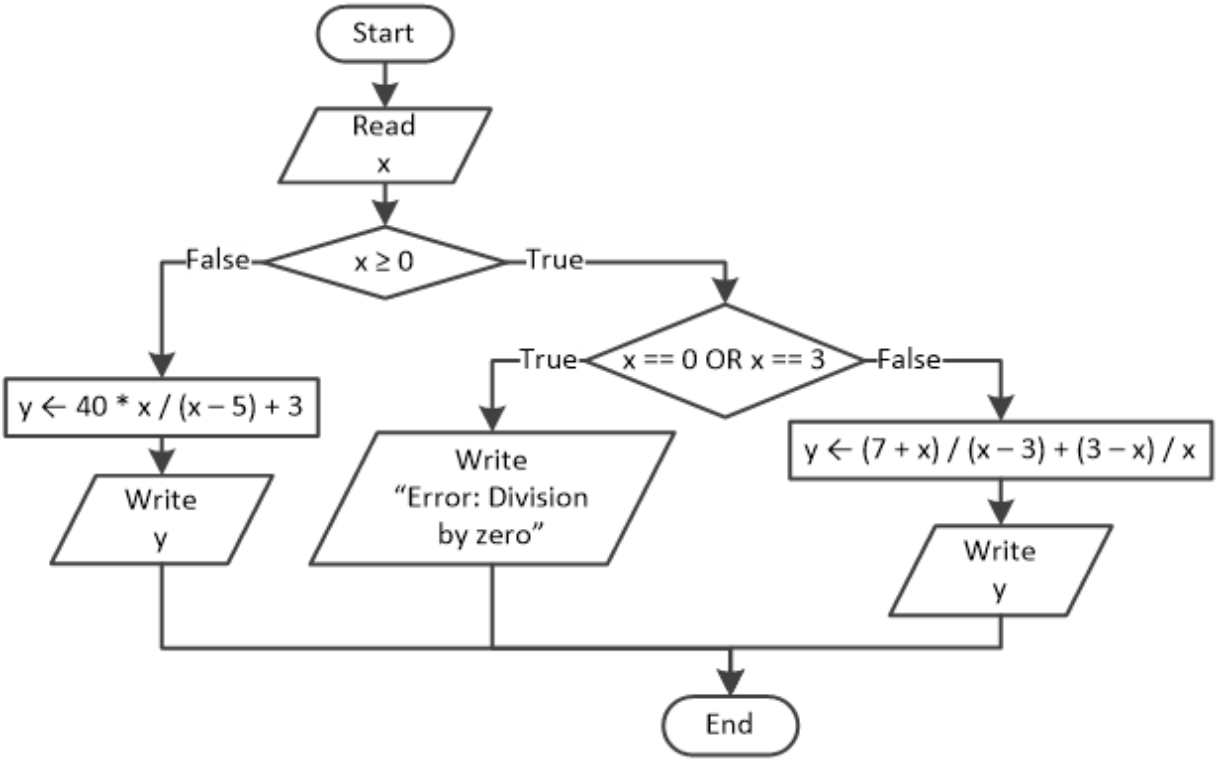
$$y = \begin{cases} \dfrac{7+x}{x-3} + \dfrac{3-x}{x}, & x \geq 0 \\ \dfrac{40x}{x-5} + 3, & x < 0 \end{cases}$$

Solution

The formula has two different results.

- When *x* is greater than or equal to zero, the value of *y* in $\frac{7+x}{x-3} + \frac{3x}{x}$ can be found following the method shown in the previous exercise.
- However, for an *x* less than zero, a small detail can save you some lines of code. Upon closer examination, it's evident that there are no restrictions on the fraction $\frac{40x}{x-5}$ because *x* can never be +5; thus, the denominator will never be zero. This is because in the given formula *x* is less than zero!

The flowchart is shown here.



The Java program is shown here.

Project_23.3-2

```
public static void main(String[] args) throws Exception {
   double x, y;
   x = Double.parseDouble(cin.nextLine()); if (x >= 0 ) {
      if (x == 0 || x == 3) {
         System.out.println("Error: Division by zero!");
      }
      else {
         y = (7 + x) / (x - 3) + (3 - x) / x;
         System.out.println(y);
      }
   }
   else {
      y = 40 * x / (x - 5) + 3;
      System.out.println(y);
   }
}
```

Exercise 23.3-3 Solving the Linear Equation ax + b = 0

Design a flowchart and write the corresponding Java program that finds and displays the root of the linear equation `ax + b = 0`

Solution

In the equation $ax + b = 0$, the coefficients a and b are known real numbers, and x represents an unknown quantity to be found. Because x is raised to the first power, this equation is classified as a *first-degree equation*, also known as a *linear equation*.

The *root of the equation* is the value of x, for which this equation is satisfied; that is, the left side of the equality $ax + b$ equals zero.

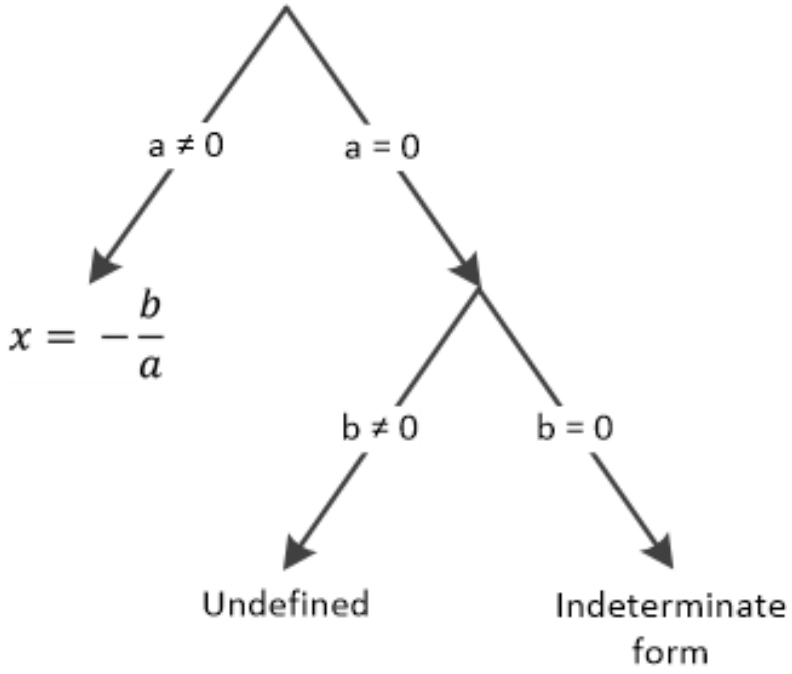
In this exercise, the user must enter values for coefficients a and b, and the program must find the value of x for which $ax + b$ equals zero.

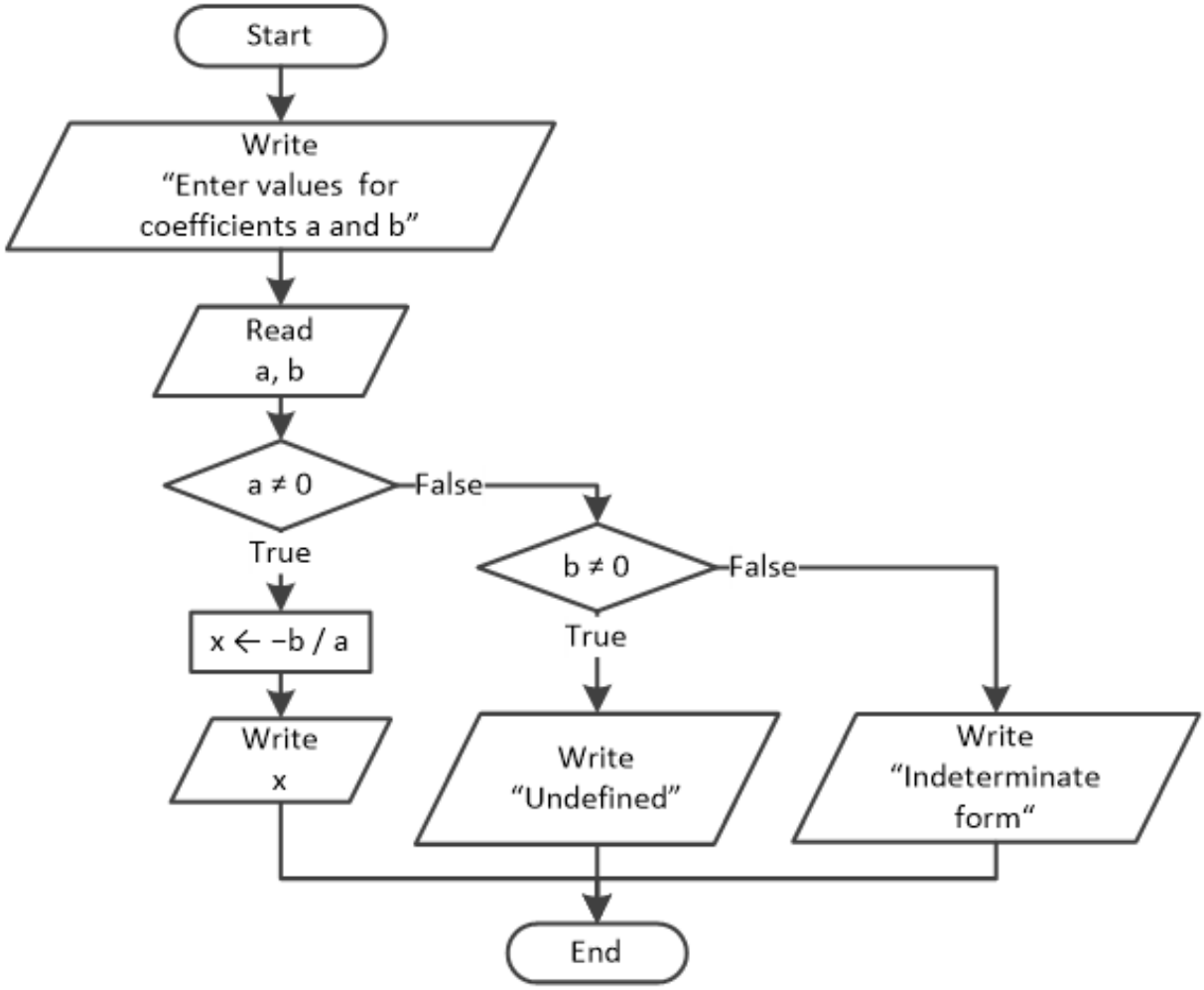
The equation $ax + b = 0$, when solved for x, becomes $x = -b / a$. Depending on the user's entered data, three possible situations can arise: i) The user might enter the value 0 for coefficient a and a non-zero value for coefficient b. In this situation, the result of $x = -b / a$ is undefined. The division by zero, as you already know from mathematics, cannot be performed.

ii) The user might enter the value 0 for both coefficients a and b. In this situation, the result of $x = -b / a$ has no defined value, and it is called an indeterminate form.

iii) The user might enter any other pair of values.

These three situations and the resulting paths are represented below with the use of a multiple-alternative decision structure.





The Java program is shown here.

Project_23.3-3

```
public static void main(String[] args) throws Exception {
   double a, b, x;
   System.out.println("Enter values for coefficients a and b: "); a =
   Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine());
   if (a != 0) {
      x = -b / a;
      System.out.println(x);
   }
   else if (b != 0) {
      System.out.println("Undefined");
   }
   else {
      System.out.println("Indeterminate form");
   }
}
```

Exercise 23.3-4 Solving the Quadratic Equation $ax^2 + bx + c = 0$

Design a flowchart and write the corresponding Java program that finds and displays the roots of the quadratic equation $ax^2 + bx + c = 0$

Solution

In the equation $ax^2 + bx + c = 0$, the coefficients a, b, and c are known real numbers, and x represents an unknown quantity to be found. Because x is raised to the second power, this equation is classified as a *second-degree equation*, also known as a *quadratic equation.*

The *roots of the equation* are the values of x, for which this equation is satisfied; that is, the left side of the equality $ax^2 + bx + c$ equals zero.

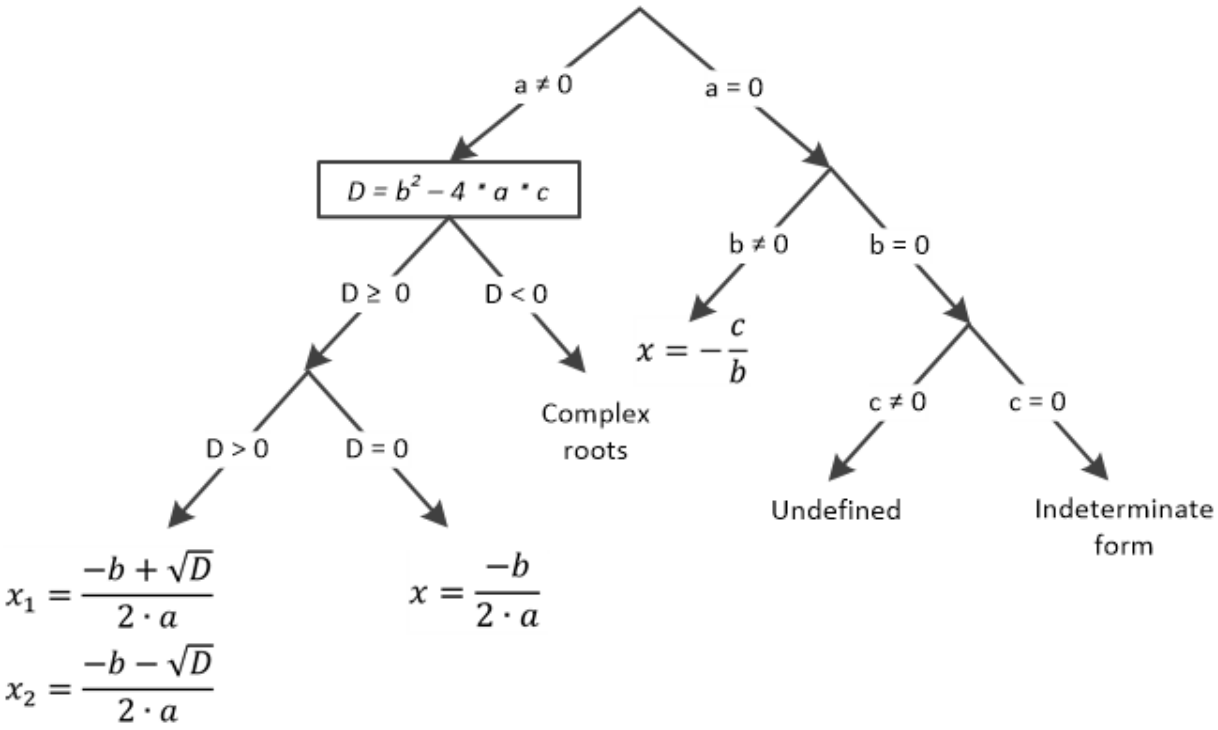
In this exercise, the user must enter values for coefficients a, b, and c, and the program must find the value(s) of x for which $ax^2 + bx + c$ equals zero.

This problem can be divided into two individual subproblems depending on the value of coefficient a.

i) If coefficient a is not equal to zero, the roots of the equation can be found using the discriminant D. Please note that the solution to this exercise presented below finds no complex roots when $D < 0$; this is beyond the scope of this book.

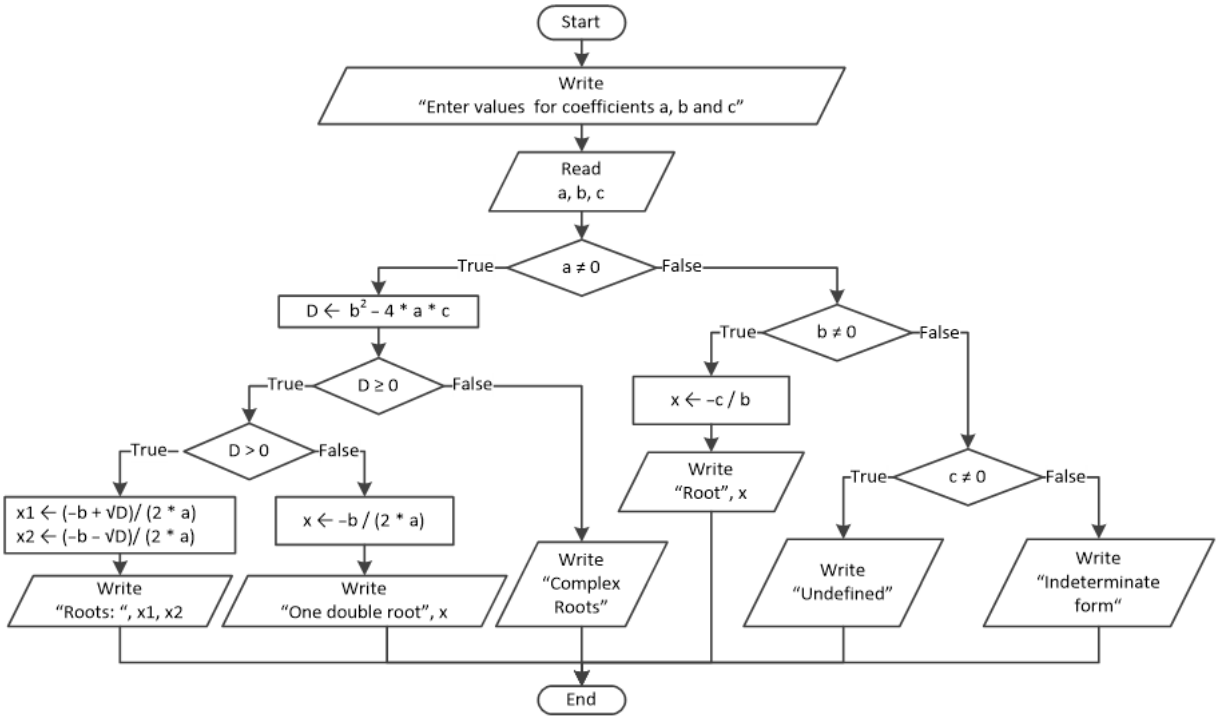
ii) If coefficient a is equal to zero, the equation becomes a linear equation, $bx + c = 0$, for which the solution was provided in the previous exercise (Exercise 23.3-3).

All necessary paths are shown here.



✎ *The path on the right* (`a = 0`) *is the solution to the linear equation* `bx + c = 0`.

Using this diagram you can design the following flowchart.


Start
Write "Enter values for coefficients a, b and c"
Read a, b, c
a ≠ 0
True
False
D ← b² – 4 * a * c
b ≠ 0
D ≥ 0
x ← –c / b
D > 0
Write "Root", x
c ≠ 0
x1 ← (–b + √D)/ (2 * a)
x2 ← (–b – √D)/ (2 * a)
x ← –b / (2 * a)
Write "Roots: ", x1, x2
Write "One double root", x
Write "Complex Roots"
Write "Undefined"
Write "Indeterminate form"
End

The Java program is shown here.

Project_23.3-4

```
public static void main(String[] args) throws Exception {
   double a, b, c, D, x1, x2, x;
   System.out.println("Enter values for coefficients a, b and c: "); a =
   Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine()); c =
   Double.parseDouble(cin.nextLine());
   if (a != 0) {
      D = b * b - 4 * a * c;
      if (D >= 0) {
         if (D > 0) {
            x1 = (-b + Math.sqrt(D)) / (2 * a);
            x2 = (-b - Math.sqrt(D)) / (2 * a);
            System.out.println("Roots: " + x1 + ", " + x2);
         }
         else {
            x = -b / (2 * a);
            System.out.println("One double root: " + x);
         }
      }
      else {
         System.out.println("Complex Roots");
      }
   }
   else {
      if (b != 0) {
         x = -c / b;
         System.out.println("Root: " + x);
      }
      else if (c != 0) {
         System.out.println("Undefined");
      }
      else {
         System.out.println("Indeterminate form");
      }
   }
}
```

23.4 Exercises with Series of Consecutive Ranges of Values

As you have already seen, in many problems the value of a variable or the result of an expression can define which statement or block of statements must be executed. In the exercises that follow, you will learn how to test if a value or the result of an expression belongs within a specific range of values (from a series of consecutive ranges of values).

Suppose that you want to display a message indicating the types of clothes a woman might wear at different temperatures.

Outdoor Temperature (in degrees Fahrenheit)	Types of Clothes a Woman Might Wear
Temperature < 45	Sweater, coat, jeans, shirt, shoes
45 ≤ Temperature < 65	Sweater, jeans, jacket, shoes
65 ≤ Temperature < 75	Capris, shorts, t-shirt, tank top, flip flops, athletic shoes
75 ≤ Temperature	Shorts, t-shirt, tank top, skort, skirt, flip flops

At first glance, single-alternative decision structures might seem like the logical choice. While not incorrect, a more in-depth analysis reveals that each condition is interdependent, meaning that when one of these evaluates to `true`, none of the others should be evaluated. You need to select just one alternative from a set of possibilities.

To solve this type of exercise, you can use a multiple-alternative decision structure or nested decision control structures. However, the former is the best choice, as it is more convenient and increases readability, as you can see in the code fragment that follows.

```
if (temperature < 45) System.out.println("Sweater, coat, jeans, shirt, shoes"); else if (temperature >= 45 && temperature < 65) System.out.println("Sweater, jeans, jacket, shoes"); else if (temperature >= 65 && temperature < 75) System.out.println("Capris, shorts, t-shirt, tank top, flip flops, athletic shoes"); else if (temperature >= 75) System.out.println("Shorts, t-shirt, tank top, skort, skirt, flip flops");
```

However, upon closer examination, it becomes apparent that all the underlined Boolean expressions are not actually required. For example, if the first Boolean expression (`temperature < 45`) evaluates to `false`, the flow of execution continues to evaluate the second Boolean expression. In this step, however, variable `temperature` is definitely greater than or equal to 45 because of the first Boolean expression, which has already evaluated to `false`. Therefore, the Boolean expression `temperature >= 45`, when evaluated, is certainly `true` and thus can be omitted. The same logic applies to all cases; you can omit all the underlined Boolean expressions. The final code fragment is shown here, with all unnecessary evaluations removed.

```
if (temperature < 45) System.out.println("Sweater, coat, jeans, shirt, shoes"); else if
(temperature < 65) System.out.println("Sweater, jeans, jacket, shoes"); else if
(temperature < 75) System.out.println("Capris, shorts, t-shirt, tank top, flip flops,
athletic shoes"); else
   System.out.println("Shorts, t-shirt, tank top, skort, skirt, flip flops");
```

Exercise 23.4-1 Calculating the Discount

A customer receives a discount based on the total amount of their order. If the total amount ordered is less than $30, no discount is given. If the total amount is equal to or greater than $30 and less than $70, a discount of 5% is applied. If the total amount is equal to or greater than $70 and less than $150, a discount of 10% is applied. If the total amount is $150 or more, the customer receives a discount of 20%. Write a Java program that prompts the user to enter the total amount of their order and then calculates and displays the applied discount rate, the discount amount in dollars, and the final after-discount amount. Assume that the user enters a non-negative value for the amount.

Solution

The following table summarizes the various discounts that are offered.

Range	Discount
amount < $30	0%
$30 ≤ amount < $70	5%
$70 ≤ amount < $150	10%
$150 ≤ amount	20%

The Java program is as follows.

```
Project_23.4-1a
public static void main(String[] args) throws Exception {
   double amount, discountAmount, finalAmount, discount =
                                0;
     System.out.print("Enter total amount: "); amount =
            Double.parseDouble(cin.nextLine());
                  if (amount < 30) {
                     discount = 0;
                  }
```

```
        else if (amount >= 30 && amount < 70) {
                discount = 5;
                }
        else if (amount >= 70 && amount < 150) {
                discount = 10;
                }
              else if (amount >= 150) {
                discount = 20;
                }
   discountAmount = amount * discount / 100; finalAmount
              = amount - discountAmount;
   System.out.println("You got a discount of " + discount
        + "%"); System.out.println("You saved $" +
   discountAmount); System.out.println("You must pay $" +
                    finalAmount); }
```

However, since it is given that the user enters valid values and not negative ones, all the underlined Boolean expressions are not actually required. The final Java program is shown here, with all unnecessary evaluations removed.

```
                  Project_23.4-1b
public static void main(String[] args) throws Exception {
    double amount, discountAmount, finalAmount, discount;
     System.out.print("Enter total amount: "); amount =
            Double.parseDouble(cin.nextLine());
                if (amount < 30) {
                discount = 0;
                }
              else if (amount < 70) {
                discount = 5;
                }
              else if (amount < 150) {
                discount = 10;
                }
                  else {
                discount = 20;
                }
   discountAmount = amount * discount / 100; finalAmount
              = amount - discountAmount;
```

```
System.out.println("You got a discount of " + discount
      + "%"); System.out.println("You saved $" +
discountAmount); System.out.println("You must pay $" +
                        finalAmount); }
```

Exercise 23.4-2 Validating Data Input and Calculating the Discount

Rewrite the Java program of the previous exercise to validate the data input. An error message must be displayed when the user enters a negative value.

Solution

The Java program that solves this exercise, given in general form, is as follows.

```
                    Main Code
System.out.print("Enter total amount: "); amount =
        Double.parseDouble(cin.nextLine());
                  if (amount < 0) {
  System.out.println("Entered value is negative"); }
                       else {
    Code Fragment 1: Calculate and display the
    applied discount rate, the discount amount and
        the final after-discount amount.
                          }
```

Code Fragment 1 that follows is taken from the previous exercise (Exercise 23.4-1). It calculates and displays the applied discount rate, the discount amount in dollars, and the final after-discount amount.

Code Fragment 1

```
if (amount < 30) {
   discount = 0;
}
else if (amount < 70) {
   discount = 5;
}
else if (amount < 150) {
   discount = 10;
}
else {
   discount = 20;
```

```
}
discountAmount = amount * discount / 100; finalAmount = amount - discountAmount;
System.out.println("You got a discount of " + discount + "%"); System.out.println("You
saved $" + discountAmount); System.out.println("You must pay $" + finalAmount);
```

After embedding **Code Fragment 1** in **Main Code**, the final Java program becomes Project_23.4-2

```
public static void main(String[] args) throws Exception {
   double amount, discountAmount, finalAmount, discount;
   System.out.print("Enter total amount: "); amount =
   Double.parseDouble(cin.nextLine());
   if (amount < 0) {
      System.out.println("Entered value is negative");
   }
   else {
      if (amount < 30) { [More…]
         discount = 0;
      }
      else if (amount < 70) {
         discount = 5;
      }
      else if (amount < 150) {
         discount = 10;
      }
      else {
         discount = 20;
      }
      discountAmount = amount * discount / 100;
      finalAmount = amount - discountAmount;
      System.out.println("You got a discount of " + discount + "%");
      System.out.println("You saved $" + discountAmount);
      System.out.println("You must pay $" + finalAmount);

   }
}
```

Exercise 23.4-3 Sending a Parcel

In a post office, the shipping cost for sending a medium parcel depends on its weight and whether its destination is inside or outside the country. Shipping costs are calculated according to the following table.

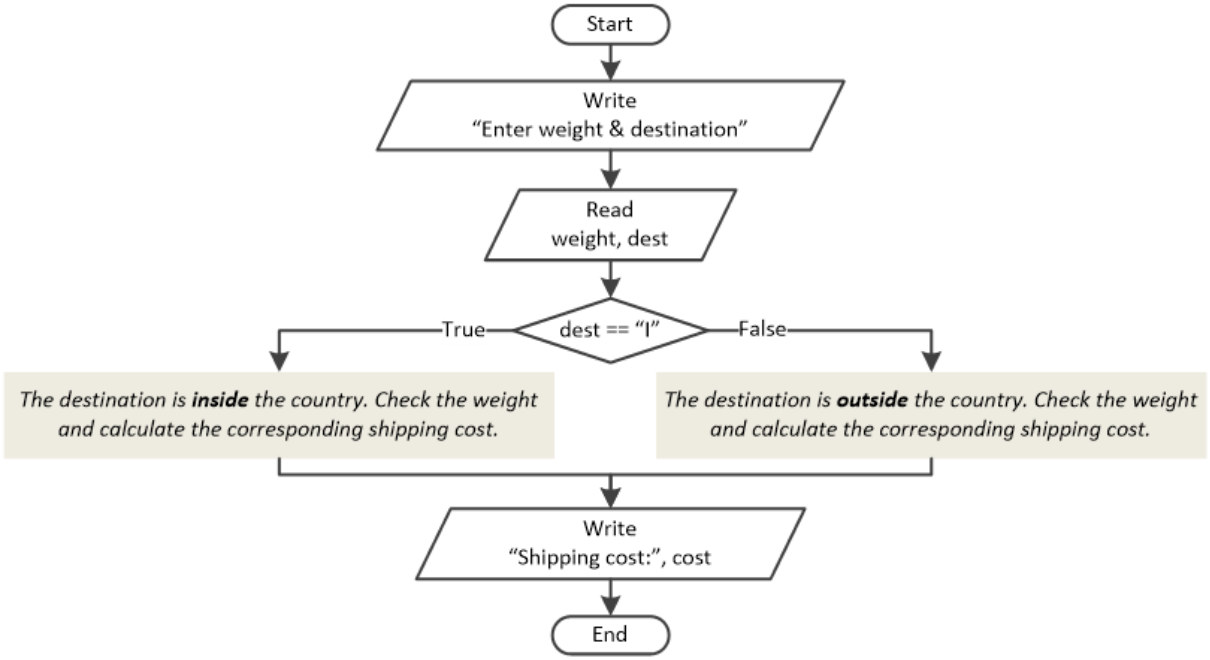
Parcel's Weight (in lb)	Destination Inside the Country (in USD per lb)	Destination Outside the Country (in USD)

weight ≤ 1	$0.010	$10
1 < weight ≤ 2	$0.013	$20
2 < weight ≤ 4	$0.015	$50
4 < weight	$0.020	$60

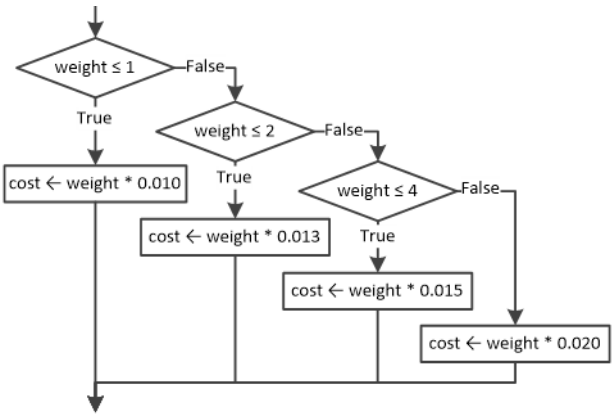
Design a flowchart and write the corresponding Java program that prompts the user to enter the weight of a parcel and its destination (I: inside the country, O: outside the country) and then calculates and displays the shipping cost.

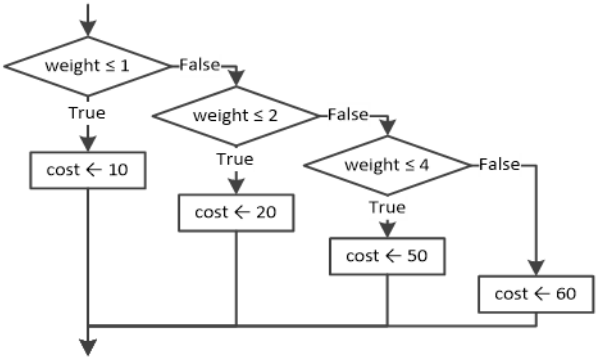
Solution

The following flowchart, given in general form, solves this exercise.

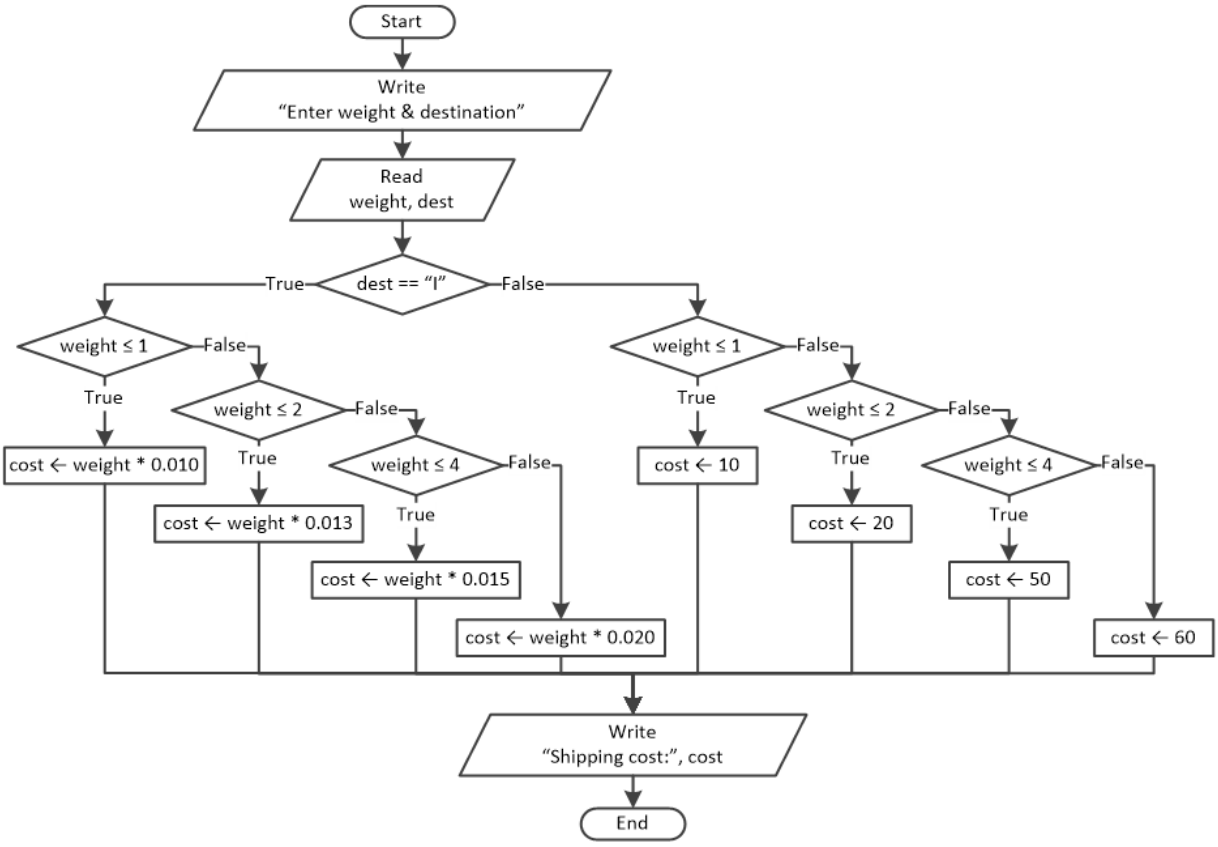


Now you need two multiple-alternative decision structures to calculate the shipping cost for parcels sent **inside** and **outside** the country. These are shown in the respective left and right flowchart fragments below.





After combining these two flowcharts with the previous one, the final flowchart becomes



The corresponding Java program is shown here.

Project_23.4-3

```
public static void main(String[] args) throws Exception {
   double weight, cost; String dest;
   System.out.print("Enter weight & destination: "); weight =
   Double.parseDouble(cin.nextLine()); dest = cin.nextLine();
   if (dest.toUpperCase().equals("I")) {
```

```
    if (weight <= 1) { [More…]
       cost = weight * 0.010;
    }
    else if (weight <= 2) {
       cost = weight * 0.013;
    }
    else if (weight <= 4) {
       cost = weight * 0.015;
    }
    else {
       cost = weight * 0.020;
    }
  }
  else {
    if (weight <= 1) { [More…]
       cost = 10;
    }
    else if (weight <= 2) {
       cost = 20;
    }
    else if (weight <= 4) {
       cost = 50;
    }
    else {
       cost = 60;
    }
  }
  System.out.println("Shipping cost: " + cost); }
```

✎ *A user may enter the letter I (for destination) in lowercase or uppercase. The method* `toUpperCase()` *ensures that the program executes properly for both cases.*

✎ *The statement* `if (dest.toUpperCase().equals("I"))` *is equivalent to the statement* `if (dest. toUpperCase().equals("I") == true)`**.**

Exercise 23.4-4 Finding the Values of y

Design a flowchart and write the corresponding Java program that finds and displays the values of y (if possible) in the following formula

$$y = \begin{cases} \dfrac{x}{x-3} + \dfrac{8+x}{x+1}, & -5 < x \le 0 \\ \dfrac{40x}{x-8}, & 0 < x \le 6 \\ \dfrac{3x}{x-9}, & 6 < x \le 20 \\ |x|, & \textit{for all other values of } x \end{cases}$$

Solution

In this exercise, there are two restrictions on the fractions: ► In fraction $\frac{8+x}{x+1}$, the value of *x* cannot be −1.

► In fraction $\frac{3x}{x-9}$, the value of *x* cannot be +9.

For all other fractions, it's impossible for the denominators to be set to zero because of the range in which x belongs.

The Java program is shown here.

```
Project_23.4-4a
public static void main(String[] args) throws Exception {
                    double x, y;
      System.out.print("Enter a value for x: "); x =
           Double.parseDouble(cin.nextLine());
                 if (x > -5 && x <= 0) {
            if (x != -1) {  [More…]
          y = x / (x - 3) + (8 + x) / (x + 1);
                System.out.println(y);
                          }
                       else {
          System.out.println("Invalid value");
                          }
                    }
              else if (x > 0 && x <= 6) {
```

```
            y = 40 * x / (x - 8);  [More…]
            System.out.println(y);
        }
    else if (x > 6 && x <= 20) {
        if (x != 9) {  [More…]
            y = 3 * x / (x - 9);
            System.out.println(y);
        }
        else {
            System.out.println("Invalid value");
        }
    }
    else {
        y = Math.abs(x);  [More…]
        System.out.println(y);
    }
}
```

If you are wondering whether you can remove all `System.out.println(y)` statements and instead have a single `System.out.println(y)` statement at the end of the program, the answer is “no”. Since there are paths that do not include that statement, you must include it in every required path. However, by making a slight modification to the code and checking for invalid values at the beginning, you can have the opportunity to move the `System.out.println(y)` statement to the end of all paths. The modified Java program is shown here.

```
Project_23.4-4b
public static void main(String[] args) throws Exception {
    double x, y;
    System.out.print("Enter a value for x: "); x =
        Double.parseDouble(cin.nextLine());
    if (x == -1 || x == 9) {
        System.out.println("Invalid value");
    }
    else {
        if (x > -5 && x <= 0) {
            y = x / (x - 3) + (8 + x) / (x + 1);
```

```
                }
            else if (x > 0 && x <= 6) {
                y = 40 * x / (x - 8);
                }
            else if (x > 6 && x <= 20) {
                y = 3 * x / (x - 9);
                }
                else {
                y = Math.abs(x);
                }
            System.out.println(y);
                }
            }
```

Now, you might be wondering if the underlined Boolean expressions are redundant, right? Suppose you do remove them, and the user enters a value of −20 for `x`. The flow of execution would then reach the Boolean expression `x <= 0`, which would evaluate to `true`. This means that the fraction $\frac{x}{x-3} + \frac{8+x}{x+1}$ would be calculated instead of the absolute value of x.

To be able to remove the underlined Boolean expressions, you need to make a slight modification to the code. The key here is to first examine the case of the absolute value of x. Following that, you can find a proposed solution below.

```
                🗀 Project_23.4-4c
public static void main(String[] args) throws Exception {
                    double x, y;
        System.out.print("Enter a value for x: "); x =
            Double.parseDouble(cin.nextLine());
                if (x == -1 || x == 9) {
            System.out.println("Invalid value");
                    }
                else {
            if (x <= -5 || x > 20) {
                y = Math.abs(x);
                }
            else if (x <= 0) {
        y = x / (x - 3) + (8 + x) / (x + 1);
```

```
                }
            else if (x <= 6) {
            y = 40 * x / (x - 8);
                }
                else {
             y = 3 * x / (x - 9);
                }
        System.out.println(y);
                }
            }
```

✎ *It is obvious that one problem can have many solutions. It is up to you to find the optimal one!*

Exercise 23.4-5 Progressive Rates and Electricity Consumption

The LAV Electricity Company charges subscribers for their electricity consumption according to the following table (monthly rates for domestic accounts). Assume that all extra charges such as transmission service charges and distribution charges are all included.

Kilowatt-hours (kWh)	USD per kWh
kWh ≤ 500	$0.10
501 ≤ kWh ≤ 2000	$0.25
2001 ≤ kWh ≤ 4500	$0.40
4501 ≤ kWh	$0.60

Write a Java program that prompts the user to enter the total number of kWh consumed and then calculates and displays the total amount to pay.

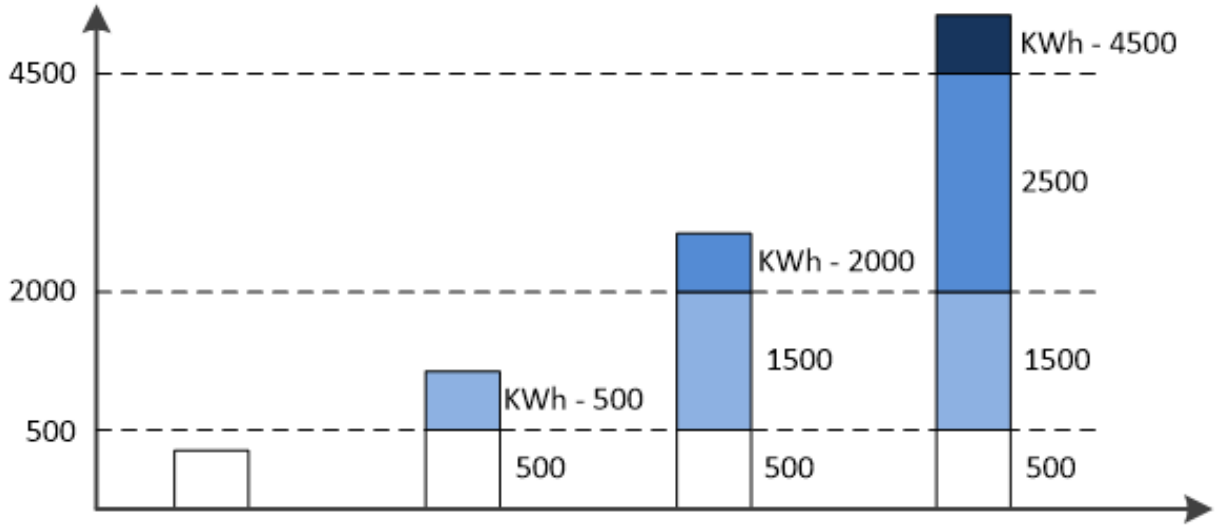
Please note that the rates are progressive.

Solution

The term *progressive rates* means that when a customer consumes, for example, 2200 kWh, not all of the kilowatt-hours are charged at $0.40. The first 500 kWh are charged at $0.10, the next 1500 kWh are charged at $0.25 and only the last 200 kWh are charged at $0.40. Thus, the customer must pay 500 × $0.10 + 1500 × $0.25 + 200 × $0.40 = $505

Applying the same logic, the total amount to be paid when the customer consumes, say, 4800 kWh can be calculated as follows. The first 500 kWh are charged at $0.10, the next 1500 kWh are charged at $0.25, the next 2500 kWh are charged at 0.40, and only the last 300 kWh are charged at $0.60. Thus, the customer must pay 500 × $0.10 + 1500 × $0.25 + 2500 × $0.40 + 300 × $0.60 = $1605

The following diagram can help you fully understand how to calculate the total amount to pay when the rates are progressive.



The Java program is shown here.

Project_23.4-5

```
public static void main(String[] args) throws Exception {
   int kwh; double t;
   System.out.print("Enter number of Kilowatt-hours consumed: "); kwh =
   Integer.parseInt(cin.nextLine());
   if (kwh <= 500) {
      t = kwh * 0.10;
   }
   else if (kwh <= 2000) {
      t = 500 * 0.10 + (kwh - 500) * 0.25;
   }
   else if (kwh <= 4500) {
      t = 500 * 0.10 + 1500 * 0.25 + (kwh - 2000) * 0.40;
   }
   else {
      t = 500 * 0.10 + 1500 * 0.25 + 2500 * 0.4 + (kwh - 4500) * 0.60;
   }
   System.out.println("Total amount to pay: " + t); }
```

Exercise 23.4-6 Progressive Rates and Text Messaging Services

The LAV Cell Phone Company charges customers a basic rate of $8 per month to send text messages. Additional rates are charged based on the total number of text messages sent, as shown in the following table.

Number of Text Messages Sent	USD per text message
Up to 50	Free of charge
51 - 150	$0.05
151 and above	$0.10

Federal, state, and local taxes add a total of 10% to each bill.

Write a Java program that prompts the user to enter the number of text messages sent and then calculates and displays the total amount to pay.

Please note that the rates are progressive.

Solution

The Java program is presented here.

Project_23.4-6

```
public static void main(String[] args) throws Exception {
   int count; double extra, totalWithoutTaxes, taxes, total;
   System.out.print("Enter number of text messages sent: "); count =
   Integer.parseInt(cin.nextLine());
   if (count <= 50) {
      extra = 0;
   }
   else if (count <= 150) {
      extra = (count - 50) * 0.05;
   }
   else {
      extra = 100 * 0.05 + (count - 150) * 0.10;
   }
   totalWithoutTaxes = 8 + extra; //Add basic rate of $8
   taxes = totalWithoutTaxes * 10 / 100; //Calculate the total taxes total =
   totalWithoutTaxes + taxes; //Calculate the total amount to pay
   System.out.println("Total amount to pay: " + total); }
```

23.5 Exercises of a General Nature with Decision Control Structures

Exercise 23.5-1 Finding a Leap Year

Write a Java program that prompts the user to enter a year and then displays a message indicating whether it is a leap year; otherwise the message "Not a leap year" must be displayed. Moreover, if the user enters a year less than 1582, an error message must be displayed.

Solution

According to the Gregorian calendar, which was first introduced in 1582, a year is a leap year when at least one of the following conditions is met: **1st Condition**: The year is exactly divisible by 4, and not by 100.

2nd Condition: The year is exactly divisible by 400.

In the following table, some years are not leap years because neither of the two conditions evaluates to `true`.

Year	Leap Year	Conditions
1600	***Yes***	2nd Condition is `true`. It is exactly divisible by 400
1900	*No*	Both conditions are `false`.
1904	***Yes***	1st Condition is `true`. It is exactly divisible by 4, and not by 100
1905	*No*	Both conditions are `false`.
2000	***Yes***	2nd Condition is `true`. It is exactly divisible by 400
2002	*No*	Both conditions are `false`.
2004	***Yes***	1st Condition is `true`. It is exactly divisible by 4, and not by 100
2024	***Yes***	1st Condition is `true`. It is exactly divisible by 4, and not by 100

The Java program is shown here.

Project_23.5-1

```
public static void main(String[] args) throws Exception {
   int y;
   System.out.print("Enter a year: "); y = Integer.parseInt(cin.nextLine());
   if (y < 1582) {
```

```
      System.out.println("Error! The year cannot be less than 1582");
   }
   else {
      if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0) {
         System.out.println("Leap year!");
      }
      else {
         System.out.println("Not a leap year");
      }
   }
}
```

The AND (`&&`) operator has a higher precedence than the OR (`||`) operator.

Exercise 23.5-2 Displaying the Days of the Month

Write a Java program that prompts the user to enter a year and a month and then displays how many days are in that month. The program needs to take into consideration the leap years. In case of a leap year, February has 29 instead of 28 days. Moreover, if the user enters a year less than 1582, an error message must be displayed.

Solution

The following Java program, given in general form, solves this exercise.

```
Main Code
public static void main(String[] args) throws Exception {
                        int m, y;
          System.out.print("Enter a year: "); y =
             Integer.parseInt(cin.nextLine());
                     if (y < 1582) {
       System.out.println("Error! The year cannot be less
                        than 1582");
                           }
                         else {
         System.out.print("Enter a month (1 – 12): ");
             m = Integer.parseInt(cin.nextLine());
                     if (m == 2) {
```

```
        Code Fragment 1: Check whether the year (in
        variable y) is a leap year and display how
                    many days are in February.
                                }
        else if (m == 4 || m == 6 || m == 9 || m == 11) {
            System.out.println("This month has 30 days");
                                }
                            else {
            System.out.println("This month has 31 days");
                                }
                             }
                            }
```

Code Fragment 1, shown here, checks whether the year (in variable y) is a leap year and displays how many days are in February.

Code Fragment 1

```
if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0) {
   System.out.println("This month has 29 days"); }
else {
   System.out.println("This month has 28 days"); }
```

After embedding **Code Fragment 1** in **Main Code**, the final Java program becomes Project_23.5-2a

```
public static void main(String[] args) throws Exception {
    int m, y;
    System.out.print("Enter a year: "); y = Integer.parseInt(cin.nextLine());
    if (y < 1582) {
        System.out.println("Error! The year cannot be less than 1582");
    }
    else {
        System.out.print("Enter a month (1 – 12): ");
        m = Integer.parseInt(cin.nextLine());
        if (m == 2) {
```

```
            if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0) {  [More…]
                System.out.println("This month has 29 days");
            }
            else {
                System.out.println("This month has 28 days");
            }
        }
        else if (m == 4 || m == 6 || m == 9 || m == 11) {
            System.out.println("This month has 30 days");
        }
        else {
            System.out.println("This month has 31 days");
        }
    }
}
```

Below, the same problem is solved again, using, however, the case decision structure.

```
                    Project_23.5-2b
public static void main(String[] args) throws Exception {
                        int m, y;
          System.out.print("Enter year: "); y =
            Integer.parseInt(cin.nextLine());
                    if (y < 1582) {
      System.out.println("Error! The year cannot be less
                        than 1582");
                        }
                      else {
        System.out.print("Enter a month (1 – 12): ");
            m = Integer.parseInt(cin.nextLine());
                        switch (m) {
                            case 2:
            if (y % 4 == 0 && y % 100 != 0 || y % 400 ==
                                0) {
                System.out.println("This month has 29
                                days");
                              }
                            else {
```

```
                System.out.println("This month has 28
                                    days");
                                  }
                                break;
                              case 4:
                              case 6:
                              case 9:
                              case 11:
                System.out.println("This month has 30 days");
                                break;
                              default:
                System.out.println("This month has 31 days");
                                }
                              }
                            }
```

✎ *Note the way cases 4, 6, and 9 are written. Since there isn't any* `break` *statement in any of those cases, they all reach case 11.*

Exercise 23.5-3 Checking for Proper Capitalization and Punctuation

Write a Java program that prompts the user to enter a sentence and then checks it for proper capitalization and punctuation. The program must determine if the string begins with an uppercase letter and ends with a punctuation mark (check only for periods, question marks, and exclamation marks).

Solution

In this exercise you need to isolate the first and the last character of the string. As you already know, you can access any individual character of a string using substring notation. You can use index 0 to access the first character, index 1 to access the second character, and so on.

Thus, you can isolate the first character of a string using the following Java statement.

```
firstChar = sentence.charAt(0);
```

On the other hand, the index of the last character is 1 less than the length of the string. You can find the length of any string using the `length()` method.

Using the following statements, you can isolate the last character of string `sentence`

length = sentence.**length**(); lastChar = sentence.**charAt**(length − 1);

or, using the more concise statement

lastChar = sentence.**charAt**(sentence.**length**() − 1);

The Java program is shown here.

Project_23.5-3

```
public static void main(String[] args) throws Exception {
   String sentence, firstChar, lastChar; boolean sentenceIsOkay;
   System.out.print("Enter a sentence: "); sentence = cin.nextLine();
   //Get first character and convert it from char to String firstChar = "" +
   sentence.charAt(0); //Get last character and convert it from char to String lastChar
   = "" + sentence.charAt(sentence.length() - 1);
   sentenceIsOkay = true;
   if (!firstChar.equals(firstChar.toUpperCase())) {
      sentenceIsOkay = false;
   }
   else if (!lastChar.equals(".") && !lastChar.equals("?") && !lastChar.equals("!")) {
      sentenceIsOkay = false;
   }
   if (sentenceIsOkay) {
      System.out.println("Sentence is okay!");
   }
}
```

In the beginning, the program assumes that the sentence is okay (`sentenceIsOkay = true`). Then, it checks for proper capitalization and proper punctuation and if it finds something wrong, it assigns the value `false` to the variable `sentenceIsOkay`.

Exercise 23.5-4 Is the Number a Palindrome?

A palindrome is a number that remains the same after reversing its digits. For example, the number 13631 is a palindrome. Write a Java program that lets the user enter a five-digit integer and tests whether or not this number is a palindrome. Moreover, a different error message for each type of input error must be displayed when the user enters a float, or any integer with either less than or more than five digits.

(Note that this involves data validation!) *Solution*

There are actually two different approaches! The first one splits the number's digits into five different variables while the second one handles the number

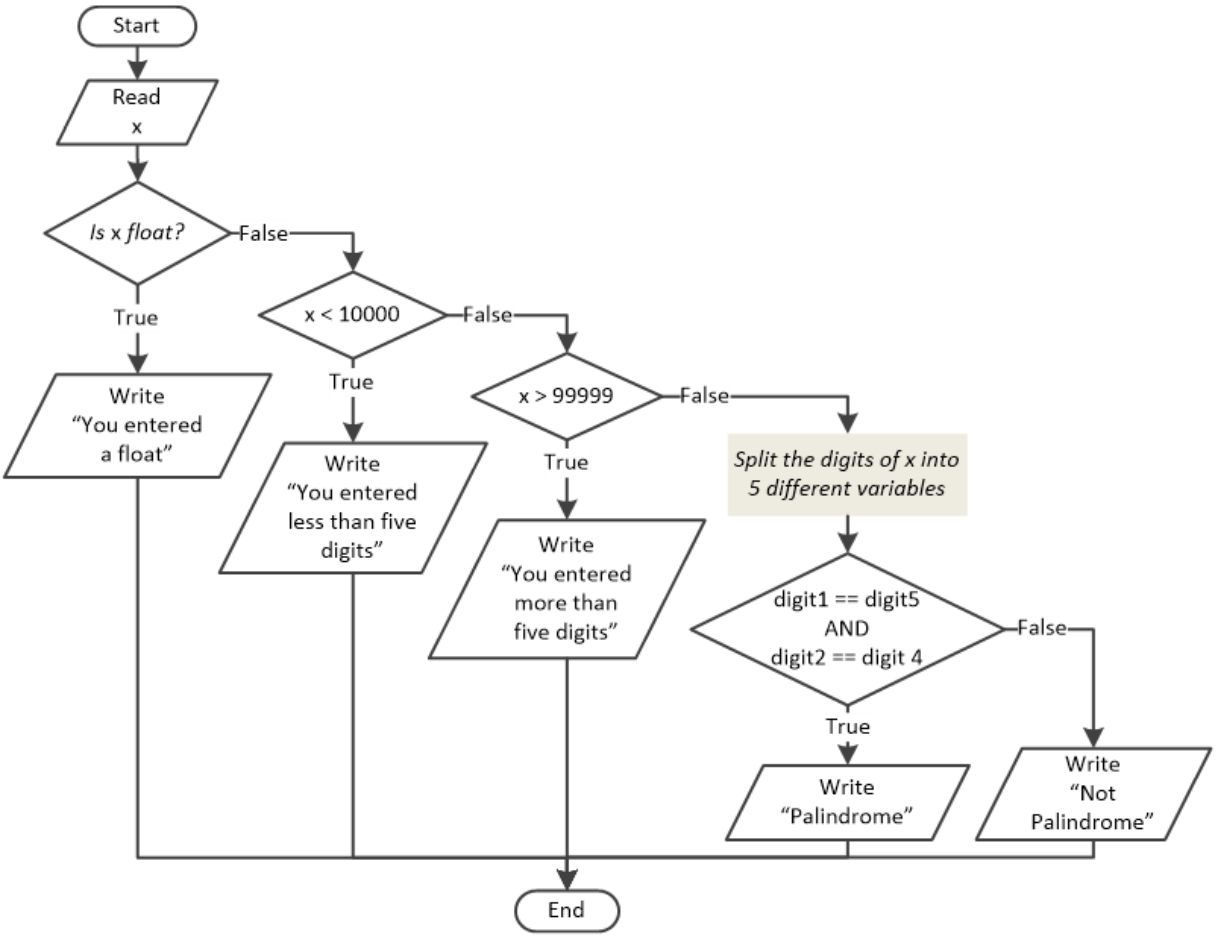
as if it were a string. Let's analyze them both!

First approach

To test if the user enters a palindrome number, you need to split its digits into five different variables as you learned in Chapter 13. Then, you can check whether the 1st digit is equal to the 5th digit and the 2nd digit is equal to the 4th digit. If this evaluates to `true`, the number is a palindrome.

To validate data input, you need to check whether the user has entered a five-digit number. Keep in mind that all five-digit numbers are in the range of 10000 to 99999. Therefore, you can just restrict the data input to within this range.

In order to display many different error messages, the best practice is to use a multiple-alternative decision structure which first checks data input validity for all cases, and then tries to solve the required problem. For example, if you need to check for various errors, you can do something like the following.



The final Java program is shown here.

```
                    Project_23.5-4a
public static void main(String[] args) throws Exception {
    int digit1, r, digit2, digit3, digit4, digit5; double
                                x;
          x = Double.parseDouble(cin.nextLine());
                     if (x != (int)x) {
          System.out.println("You entered a float");
                                }
                     else if (x < 10000) {
        System.out.println("You entered less than five
                                digits");
                                }
                     else if (x > 99999) {
        System.out.println("You entered more than five
                                digits");
                                }
                                else {
      //Split the digits of x into 5 different variables
                     digit1 = (int)(x / 10000);
                        r = (int)x % 10000;
                    digit2 = (int)(r / 1000);
                          r = r % 1000;
                    digit3 = (int)(r / 100);
                          r = r % 100;
                    digit4 = (int)(r / 10);
                        digit5 = r % 10;
          if (digit1 == digit5 && digit2 == digit4) {
                System.out.println("Palindrome");
                                }
                               else {
              System.out.println("Not Palindrome");
                                }
                               }
                              }
```

Second approach

This approach handles the number as if it were a string. It compares the first character to the last one and the second character to the last but one. If they

are equal, it means that the number is a palindrome. The Java program is shown here.

```
                    Project_23.5-4b
public static void main(String[] args) throws Exception {
                   String xStr; double x;
          x = Double.parseDouble(cin.nextLine());
                     if (x != (int)x) {
          System.out.println("You entered a float");
                              }
                    else if (x < 10000) {
        System.out.println("You entered less than five
                             digits");
                              }
                    else if (x > 99999) {
        System.out.println("You entered more than five
                             digits");
                              }
                             else {
                      xStr = "" + (int)x;
            if (xStr.charAt(0) == xStr.charAt(4) &&
              xStr.charAt(1) == xStr.charAt(3)) {
                System.out.println("Palindrome");
                              }
                            else {
             System.out.println("Not palindrome");
                              }
                            }
                           }
```

23.6 Boolean Expressions Reference and Handy Tips

This section summarizes all the Boolean expressions that you've encountered on your journey with Java so far, along with additional expressions and useful tips. Whether you're a beginner or a seasoned programmer, these expressions and tips will serve as valuable tools in your coding arsenal. Keep them close, because you never know when they'll come in handy. For some of them, two or more approaches to achieve the desired result are provided.

1) How to check if number in `x` is between `A` and `B`
   - ► `x >= A && x <= B`
2) How to check if number in `x` is not between `A` and `B`
   - ► `!(x >= A && x <= B)` `x < A || x > B`
3) How to check if `x` is either equal to `A`, `B`, or `C`
   - ► `x == A || x == B || x == C`
4) How to check if `x` is neither equal to `A`, nor `B`, nor `C`
   - ► `x != A && x != B && x != C`
   - ► `!(x == A || x == B || x == C)` 5) How to check if `x` contains an integer. Please note that variable `x` must be of type `double`.
   - ► `x == (int)x` 6) How to check if `x` contains a float.
   - ► `x != (int)x` 7) How to check if `x` contains an even number.
   - ► `x % 2 == 0`
   - ► `x % 2 != 1`
   - ► `!(x % 2 == 1)` `!(x % 2 != 0)` 8) How to check if `x` contains an odd number.
   - ► `x % 2 == 1`
   - ► `x % 2 != 0`
   - ► `!(x % 2 == 0)` `!(x % 2 != 1)` 9) How to check if `x` is an integer multiple of `y` ► `x % y == 0`
10) How to isolate the decimal part of a real number ► `x - (int)x` 11) How to isolate the first decimal digit of a real number ► `(int)(x * 10) % 10`
12) How to isolate the second decimal digit of a real number ► `(int)(x * 100) % 10`
13) How to isolate the Nth decimal digit of a real number ► `(int)(x * Math.pow(10, N)) % 10`
14) How to isolate the last digit of an integer ► `x % 10`
15) How to isolate the second to last digit of an integer ► `(int)(x / 10) % 10`
16) How to isolate the Nth to last digit of an integer ► `(int)(x / Math.pow(10, N)) % 10`
17) How to check if a word/sentence starts with the letter “B”

► `x.charAt(0) == 'B'`

18) How to check if a word/sentence ends with a period "."

► `x.charAt(x.length() – 1) == '.'`

19) How to find the middle number among three numbers `x`, `y`, and `z` ► `x + y + z - minimum – maximum`

20) How to find the sum of the two smallest numbers among three numbers `x`, `y`, and `z` ► `x + y + z - maximum`

21) How to find the sum of the two greatest numbers among three numbers `x`, `y`, and `z` ► `x + y + z - minimum`

22) How to find the sum of the three middle numbers among five numbers `x`, `y`, `z`, `w`, and `u` ► `x + y + z + w + u – minimum - maximum`

23) How to check if the distance between two numbers is greater than `NUMBER`

► `Math.abs(x - y) > NUMBER`

24) How to check if a positive integer has three digits ► `x >= 100 && x <= 999`

► `String.valueOf(x).length() == 3`

► `("" + x).length() == 3`

25) How to check if an integer has three digits ► `Math.abs(x) >= 100 && Math.abs(x) <= 999`

► `String.valueOf(Math.abs(x)).length() == 3`

► `("" + Math.abs(x)).length() == 3`

26) How to check if a positive integer has four digits and starts with 5

► `x >= 5000 && x <= 5999`

27) How to check if two numbers have the same sign ► `x > 0 && y > 0 || x < 0 && y < 0`

► `x * y > 0`

28) How to check if both numbers are either even or odd ► `x % 2 == 0 && y % 2 == 0 || x % 2 == 1 && y % 2 == 1`

► `x % 2 == y % 2`

29) How to check if exactly one of the two conditions `BE1` or `BE2` is `true`, but not both (Exclusive OR operation) ► `BE1 && !(BE2) || BE2 && !(BE1)`

30) How to check if the year in `y` is a leap year ► `y % 4 == 0 && y % 100 != 0 || y % 400 == 0`

23.7 Review Exercises

Complete the following exercises.

1) Write a Java program that prompts the user to enter a numeric value and then calculates and displays its square root. Moreover, an error message must be displayed when the user enters a negative value.
2) Design a flowchart that lets the user enter an integer and, if its last digit is equal to 5, a message “Last digit equal to 5” is displayed; otherwise, a message “Nothing special” is displayed. Moreover, if the user enters a negative value, an error message must be displayed.

   Hint: You can isolate the last digit of any integer using a modulus 10 operation.
3) Design a flowchart and write the corresponding Java program that lets the user enter two integers and then displays a message indicating whether at least one integer is odd; otherwise, a message “Nothing special” is displayed. Moreover, if the user enters negative values, an error message must be displayed.
4) Design a flowchart and write the corresponding Java program that prompts the user to enter an integer, and then displays a message indicating whether this number is even; it must display “Odd” otherwise. Moreover, a different error message for each type of input error must be displayed when the user enters a negative value or a float.
5) Design a flowchart and write the corresponding Java program that prompts the user to enter an integer and then displays a message indicating whether this number is exactly divisible by 3 and by 4; otherwise the message “NN is not what you are looking for!” must be displayed (where NN is the user-provided number). For example, 12 is exactly divisible by 3 and by 4. Moreover, an error message must be displayed when the user enters a negative value or a float.
6) Design a flowchart and write the corresponding Java program that lets the user enter two integers and then displays a message indicating whether both numbers are exactly divisible by 3 and by 4; otherwise the message “X and Y are not what you are looking for!” must be displayed (where X and Y are the user-provided numbers). Moreover, a different error message for each type of input error for each integer must be displayed when the user enters negative values or floats.
7) Write a Java program that displays the following menu: 1) Convert Kelvin to Fahrenheit 2) Convert Fahrenheit to Kelvin 3) Convert Fahrenheit to

Celsius. Convert Celsius to Fahrenheit The program must then prompts the user to enter a choice (of 1, 2, 3, or 4) and a temperature value, and subsequently calculate and display the required value. Moreover, a different error message for each type of input error must be displayed when the user enters a choice other than 1, 2, 3, or 4, or a temperature value lower than absolute zero[17].

4)

It is given that *1.8 × Kelvin = Fahrenheit + 459.67*

and

$$\frac{Celsius}{5} = \frac{Fahrenheit - 32}{9}$$

8) Write a Java program that emulates the way an electronic calculator functions. The program must first prompt the user to enter an integer, then the type of operation (+, −, *, /, DIV, MOD, POWER), and finally a second integer. Subsequently, the program must perform the chosen operation and display the result. For instance, if the user enters the values 13, +, and 2, the program must display the following message: `The result of 13 + 2 equals 15`

   Make your program accept the type of operation in all possible forms such as “Div”, “DIV”, “div”, or even “DiV”. In case of a division by zero, the message “Infinite” must be displayed.

9) Rewrite the Java program of the previous exercise to validate the data input. If the user enters an input other than +, −, *, /, DIV, MOD, POWER, an error message must be displayed.

10) Write a Java program that prompts the user to enter the names and the ages of three people and then displays the names of the youngest person and the oldest person.

11) In a song contest, each artist is scored for their performance by five judges. However, according to the rules of this contest, the total score is calculated after excluding the highest and lowest scores. Write a Java program that prompts the user to enter the name of the artist and the score they receive from each judge. The program must then display the name of the artist along with their total score.

12) Write a Java program that prompts the user to enter the ages of three people and then finds and displays the age in the middle.

13) Write a Java program that prompts the user to enter the names and the ages of three people and then displays the name of the youngest person or the oldest person, depending on which one is closer to the third age in the middle.

14) An online bookstore applies the following sales policy: Buy 3 books and pay for the 2 most expensive ones. Write a Java program that lets the user enter the prices and titles of three books. It must then display the amount the customer needs to pay, as well as the title and price of the book that was provided for free.

15) Design a flowchart and write the corresponding Java program that finds and displays the value of y (if possible) in the following formula.

$$y = \frac{5x + 3}{x - 5} + \frac{3x^2 + 2x + 2}{x + 1}$$

16) Design a flowchart and write the corresponding Java program that finds and displays the values of y (if possible) in the following formula.

$$y = \begin{cases} \dfrac{x^2}{x+1} + \dfrac{3 - \sqrt{x}}{x+2}, & x \geq 10 \\ \dfrac{40x}{x-9} + 3x, & x < 10 \end{cases}$$

17) Rewrite the Java program of Exercise 23.3-2, using a multiple-alternative decision structure.

Hint: Negate the Boolean expression `x >= 0` in the outer dual-alternative decision structure and switch its two paths.

18) Write a Java program that finds and displays the values of y (if possible) in the following formula.

$$y = \begin{cases} \dfrac{x}{\sqrt{x+30}} + \dfrac{(8+x)^2}{x+1}, & -15 < x \le -10 \\ \dfrac{|40x|}{x-8}, & -10 < x \le 0 \\ \dfrac{3x}{\sqrt{x-9}}, & 0 < x \le 25 \\ x - 1, & \text{for all other values of x} \end{cases}$$

19) A positive integer is called an Armstrong number when the sum of the cubes of its digits is equal to the number itself. The number 371 is such a number, since $3^3 + 7^3 + 1^3 = 371$. Write a Java program that lets the user enter a three-digit integer and then displays a message indicating whether or not the user-provided number is an Armstrong one. Moreover, a different error message for each type of input error must be displayed when the user enters a float or any number other than a three-digit one.

20) Write a Java program that prompts the user to enter a day (1 - 31), a month (1 - 12), and a year and then finds and displays how many days are left until the end of that month. The program must take into consideration the leap years. In the case of a leap year, February has 29 instead of 28 days.

21) Write a Java program that lets the user enter a word of six letters and then displays a message indicating whether or not every second letter is capitalized. The word “AtHeNa” is such a word, but it can be also provided as “aThEnA”.

22) An online book store sells e-books for $10 each. Quantity discounts are given according to the following table.

Quantity	Discount
3 - 5	10%
6 - 9	15%
10 - 13	20%

14 - 19	27%
20 or more	30%

Write a Java program that prompts the user to enter the total number of e-books purchased and then displays the amount of discount, and the total amount of the purchase after the discount. Assume that the user enters valid values.

23) In a supermarket, the discount that a customer receives based on the before-tax amount of their order is presented in the following table.

Range	Discount
amount < $50	0%
$50 ≤ amount < $100	1%
$100 ≤ amount < $250	2%
$250 ≤ amount	3%

Write a Java program that prompts the user to enter the before-tax amount of their order and then calculates and displays the discount amount that customers receive (if any). A VAT (Value Added Tax) of 19% must be added in the end. Moreover, an error message must be displayed when the user enters a negative value.

24) The Body Mass Index (BMI) is often used to determine whether an adult person is overweight or underweight for their height. The formula used to calculate the BMI of an adult person is $BMI = \frac{weight \cdot 703}{height^2}$

Write a Java program that prompts the user to enter their age, weight (in pounds) and height (in inches) and then displays a description according to the following table.

Body Mass Index	Description
BMI < 15	Very severely underweight

15.0 ≤ BMI < 16.0	Severely underweight
16.0 ≤ BMI < 18.5	Underweight
18.5 ≤ BMI < 25	Normal
25.0 ≤ BMI < 30.0	Overweight
30.0 ≤ BMI < 35.0	Severely overweight
35.0 ≤ BMI	Very severely overweight

The message “Invalid age” must be displayed when the user enters an age less than 18.

25) The LAV Water Company charges for subscribers' water consumption according to the following table (monthly rates for domestic accounts).

Water Consumption (cubic feet)	USD per cubic foot
consumption ≤ 10	$3
11 ≤ consumption ≤ 20	$5
21 ≤ consumption ≤ 35	$7
36 ≤ consumption	$9

Write a Java program that prompts the user to enter the total amount of water consumed (in cubic feet) and then calculates and displays the total amount to pay. Please note that the rates are progressive. Federal, state, and local taxes add a total of 10% to each bill. Moreover, an error message must be displayed when the user enters a negative value.

26) Write a Java program that prompts the user to enter their taxable income and the number of their children and then calculates the total tax to pay according to the following table. However, total tax is reduced by 2% when the user has at least one child. Please note that the rates are progressive.

Taxable Income (USD)	Tax Rate

income ≤ 8000	10%
8000 < income ≤ 30000	15%
30000 < income ≤ 70000	25%
70000 < income	30%

27) The Beaufort scale is an empirical measure that relates wind speed to observed conditions on land or at sea. Write a Java program that prompts the user to enter the wind speed and then displays the corresponding Beaufort number and description according to the following table. An additional message “It's Fishing Day!!!” must be displayed when wind speed is 3 Beaufort or less. Moreover, an error message must be displayed when the user enters a negative value.

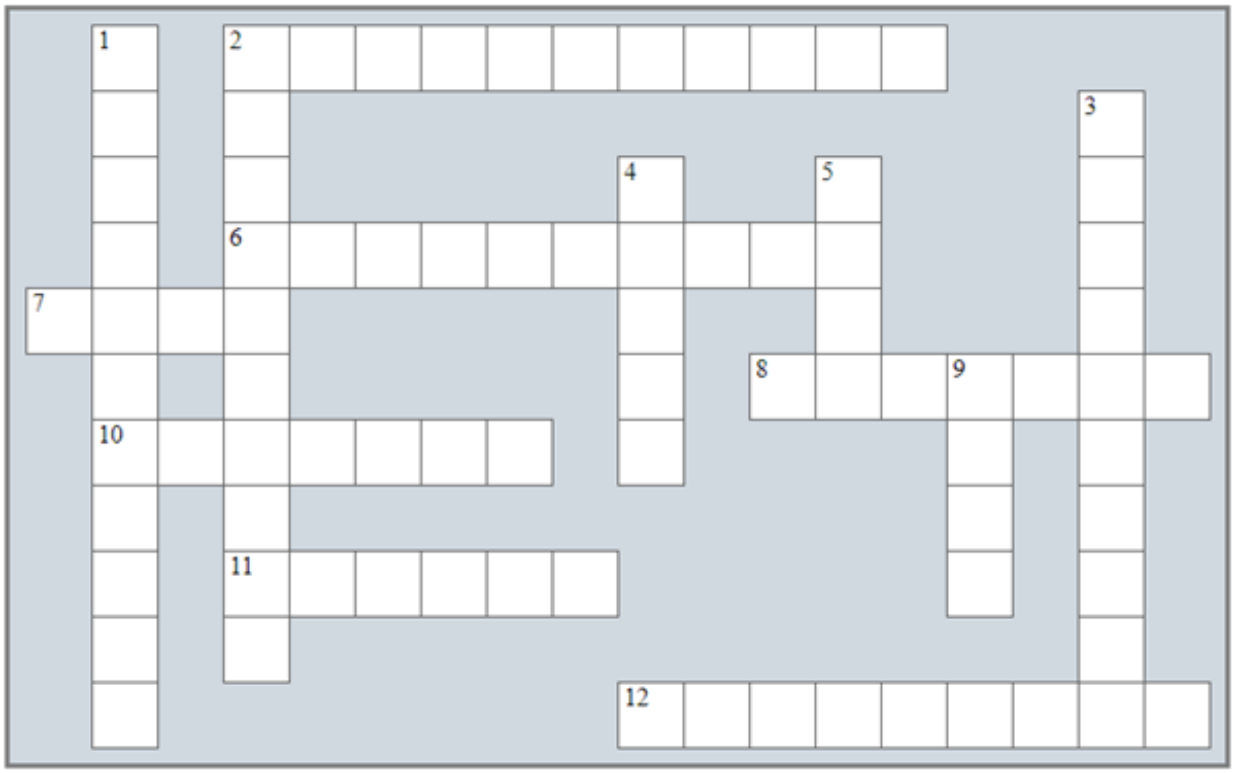
Wind Speed (miles per hour)	Beaufort Number	Description
wind speed < 1	0	Calm
1 ≤ wind speed < 4	1	Light air
4 ≤ wind speed < 8	2	Light breeze
8 ≤ wind speed < 13	3	Gentle breeze
13 ≤ wind speed < 18	4	Moderate breeze
18 ≤ wind speed < 25	5	Fresh breeze
25 ≤ wind speed < 31	6	Strong breeze
31 ≤ wind speed < 39	7	Moderate gale
39 ≤ wind speed < 47	8	Gale

47 ≤ wind speed < 55	9	Strong gale
55 ≤ wind speed < 64	10	Storm
64 ≤ wind speed < 74	11	Violent storm
74 ≤ wind speed	12	Hurricane force

Review in “Decision Control Structures”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

2) The AND (`&&`) operator is also known as a logical _____.

6) This number remains the same after reversing its digits.

7) The _____-alternative decision structure includes a statement or block of statements on both paths.

8) This is an expression that results in a value that is either `true` or `false`.

10) This Boolean expression can be built of simpler Boolean expressions.

11) This control structure is a structure that is enclosed within another structure.

12) A positive integer where the sum of the cubes of its digits is equal to the number itself.

Down

1) The OR (`||`) operator is also known as a logical _____.

2) The NOT (`!`) operator is also known as a logical _____.

3) The (>) is a _____ operator.

4) This table shows the result of a logical operation between two or more Boolean expressions for all their possible combinations of values.

5) This number is considered an even number.

9) This year is exactly divisible by 4 and not by 100, or it is exactly divisible by 400.

Review Questions

Answer the following questions.

1) What is a Boolean expression?
2) Which comparison operators does Java support?
3) Which logical operator performs a logical conjunction?
4) Which logical operator performs a logical disjunction?
5) When does the logical operator AND (`&&`) return a result of `true`?
6) When does the logical operator OR (`||`) return a result of `true`?
7) State the order of precedence of logical operators.
8) State the order of precedence of arithmetic, comparison, membership, and logical operators.
9) What is code indentation?
10) Design the flowchart and write the corresponding Java statement (in general form) of a single-alternative decision structure. Describe how this decision structure operates.
11) Design the flowchart and write the corresponding Java statement (in general form) of a dual-alternative decision structure. Describe how this decision structure operates.
12) Design the flowchart and write the corresponding Java statement (in general form) of a multiple-alternative decision structure. Describe how this decision structure operates.

13) Write the Java statement (in general form) of a case decision structure. Describe how this decision structure operates.
14) What does the term “nesting a decision structure” mean?
15) How deep can the nesting of decision control structures go? Is there any practical limit?
16) Create a diagram that shows all possible paths for solving a linear equation.
17) Create a diagram that shows all possible paths for solving a quadratic equation.
18) When is a year considered a leap year?
19) What is a palindrome number?

Part V
Loop Control Structures

Chapter 24
Introduction to Loop Control Structures

24.1 What is a Loop Control Structure?

A loop control structure is a control structure that allows the execution of a statement or block of statements multiple times until a specified condition is met.

24.2 From Sequence Control to Loop Control Structures

The next example lets the user enter four numbers and it then calculates and displays their sum. As you can see, there is no loop control structure employed yet, only the familiar sequence control structure.

```
double x, y, z, w, total;
x = Double.parseDouble(cin.nextLine()); y = Double.parseDouble(cin.nextLine()); z =
Double.parseDouble(cin.nextLine()); w = Double.parseDouble(cin.nextLine());
total = x + y + z + w;
System.out.println(total);
```

While this code is quite short, consider a similar one that allows the user to enter 1000 numbers instead of just four. Can you imagine having to write the input statement `Double.parseDouble(cin.nextLine())` a thousand times? It would certainly be more convenient if you could write this statement just once and instruct the computer to execute it a thousand times, wouldn't it? This is where a loop control structure comes into play!

But before you delve into loop control structures, try to solve a riddle first! Without using a loop control structure yet, try to rewrite the previous example, using only two variables, `x` and `total`. Yes, you heard that right! This code must calculate and display the sum of four user-provided numbers, <u>but</u> it must do so with only two variables! Can you find a way?

Hmmm… it's obvious what you are thinking right now: "*The only thing that I can do with two variables is to read one single value in variable* `x` *and then assign that value to variable* `total`". Your thinking is quite correct, and it is presented here.

```
x = Double.parseDouble(cin.nextLine()); //Read the first number total = x;
```

which can equivalently be written as

total = 0;

x = Double.**parseDouble**(cin.**nextLine**()); //*Read the first number* total = total + x;

And now what? Now, there are three things that you can actually do, and these are: think, think, and of course, think!

The first user-provided number has been stored in variable `total`, so variable `x` is now free for further use! Thus, you can reuse variable `x` to read a second value which will also accumulate in variable `total`, as follows.

```
total = 0;
x = Double.parseDouble(cin.nextLine()); //Read the first number total = total + x;
x = Double.parseDouble(cin.nextLine()); //Read the second number total = total + x;
```

✎ *Statement* `total = total + x` *accumulates the value of* `x` *in* `total`, *which means that it adds the value of* `x` *to* `total` *along with any previous value in* `total`. *For example, if variable* `total` *contains the value 5 and variable* `x` *contains the value 3, the statement* `total = total + x` *assigns the value 8 to variable* `total`.

Since the second user-provided number has been accumulated in the variable `total`, variable `x` can be reused! This process can repeat until all four numbers are read and accumulated in variable `total`. The final code is as follows. Please note that it does not use any loop control structure yet!

```
total = 0;
x = Double.parseDouble(cin.nextLine()); total = total + x;
x = Double.parseDouble(cin.nextLine()); total = total + x;
x = Double.parseDouble(cin.nextLine()); total = total + x;
x = Double.parseDouble(cin.nextLine()); total = total + x;
System.out.println(total);
```

✎ *Both this code and the initial one at the beginning of this section are considered equivalent. The main distinction between them, however, lies in the fact that this one contains four identical pairs of statements.*

Apparently, you can use this example to read and find the sum of more than four numbers. However, writing those pairs of statements multiple times can be quite cumbersome and may lead to errors if any pair is accidentally omitted.

What you truly need here is to retain just **one** pair of statements, but use a loop control structure to execute it four times (or even 1000 times, if you wish). You can use something like the following code fragment.

```
total = 0;
execute_these_statements_4_times {
   x = Double.parseDouble(cin.nextLine()); total = total + x; }
System.out.println(total);
```

Obviously there isn't any *execute_these_statements_4_times* statement in Java. This is for demonstration purposes only, but soon enough you will learn everything about all the loop control structures that Java supports!

24.3 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A loop control structure is a structure that allows the execution of a statement or block of statements multiple times until a specified condition is met.
2) It is possible to use a sequence control structure that prompts the user to enter 1000 numbers and then calculates their sum.
3) The following code fragment accumulates the value 10 in variable `total`.

   ```
   total = 10;
   a = 0;
   total = total + a;
   ```

4) The following Java program satisfies the property of effectiveness.

   ```
   public static void main(String[] args) throws Exception {
      int a, total; a = 5; total = total + a; System.out.println(total); }
   ```

5) Both of the following two code fragments assign the value of 5 to the variable `total`.

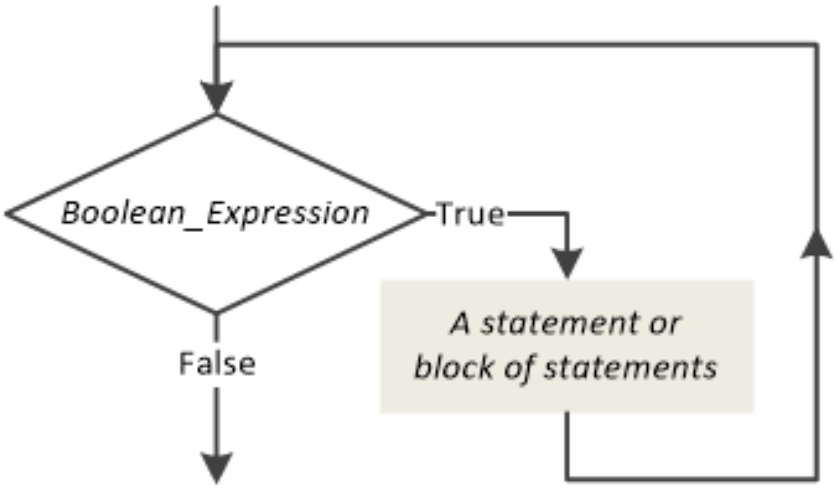
   ```
   a = 5;
   total = a;
   ```

   ```
   total = 0;
   a = 5;
   total = total + a;
   ```

Chapter 25
Pre-Test, Mid-Test and Post-Test Loop Structures

25.1 The Pre-Test Loop Structure

The pre-test loop structure is shown in the following flowchart.



Let's see what happens when the flow of execution reaches a pre-test loop structure. If *Boolean_Expression* evaluates to `true`, the statement or block of statements of the structure is executed and the flow of execution goes back to check *Boolean_Expression* once more. If *Boolean_Expression* evaluates to `true` again, the process repeats. The iterations stop when *Boolean_Expression*, at some point, evaluates to `false` and the flow of execution exits the loop.

✎ *The Decision symbol (the diamond, or rhombus) is used both in decision control structures and in loop control structures. However, in loop control structures, one of the diamond's exits always has an upward direction.*

✎ *A "pre-test loop structure" is named this way because first the Boolean expression is evaluated, and afterwards the statement or block of statements of the structure is executed.*

✎ *Because the Boolean expression is evaluated before entering the loop, a pre-test loop may perform from zero to many iterations.*

✎ *Each time the statement or block of statements of a loop control structure is executed, the term used in computer science is "the loop is iterating" or "the loop performs an iteration".*

The general form of the Java statement is

```
while (Boolean_Expression) {
    A statement or block of statements
}
```

The following example displays the numbers 1 to 10.

Project_25.1

```
public static void main(String[] args) throws Exception {
   int i;
   i = 1;
   while (i <= 10) {
      System.out.println(i);
      i++;
   }
}
```

✎ *Just as in decision control structures, the statements inside a loop control structure should be indented.*

Similar to decision control structures, when only one single statement needs to be part of the `while` statement, you are allowed to omit the braces { }. Thus, the `while` statement can be written as shown below.

```
while (Boolean_Expression)
   One_Single_Statement;
```

To prevent potential logic errors, many programmers prefer to always use braces, even when the `while` statement encloses just one statement.

Exercise 25.1-1 Designing the Flowchart and Counting the Total Number of Iterations

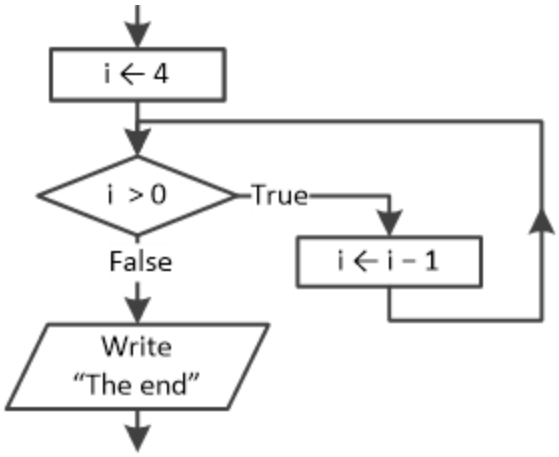
Design the corresponding flowchart for the following code fragment. How many iterations does this Java code perform?

```
int i;
i = 4;
while (i > 0) {
```

```
    i--;
}
System.out.println("The end");
```

Solution

The corresponding flowchart fragment is as follows.



Next, a trace table can help you observe the flow of execution.

Step	Statement	Notes	i	
1	`i = 4`		**4**	
2	`while (i > 0)`	This evaluates to `true`		1st iteration
3	`i--`		**3**	
4	`while (i > 0)`	This evaluates to `true`		2nd iteration
5	`i--`		**2**	
6	`while (i > 0)`	This evaluates to `true`		3rd iteration
7	`i--`		**1**	
8	`while (i > 0)`	This evaluates to `true`		4th iteration
9	`i--`		**0**	
10	`while (i > 0)`	This evaluates to `false`		
11	`.println("The end")`	It displays: The end		

As you can see from the trace table, the total number of iterations is **four**.

> ✎ *When the statement or block of statements of a pre-test loop structure is executed N times, the Boolean expression is evaluated N+1 times. Therefore, to determine the total number of iterations, count the number of times the statement or block of statements is executed, not the number of times the Boolean expression is evaluated.*

Exercise 25.1-2 Counting the Total Number of Iterations

How many iterations does this code fragment perform?

```
int i;
i = 4;
while (i >= 0) {
   System.out.println(i);
   i--;
}
System.out.println("The end");
```

Solution

This exercise is almost identical to the previous one. The main difference is that the Boolean expression here remains `true`, even for `i = 0`. Therefore, it performs an additional iteration, that is, **five** iterations.

Exercise 25.1-3 Counting the Total Number of Iterations

How many iterations does this code fragment perform?

```
int i;
i = 1;
while (i != 6) {
   i += 2;
}
System.out.println("The end");
```

Solution

Let's create a trace table to observe the flow of execution.

Step	Statement	Notes	i	
1	`i = 1`		**1**	
2	`while (i != 6)`	This evaluates to `true`		1st iteration
3	`i += 2`		**3**	

4	`while (i != 6)`	This evaluates to `true`	2nd	iteration
5	`i += 2`	**5**		
6	`while (i != 6)`	This evaluates to `true`	3rd	iteration
7	`i += 2`	**7**		
8	`while (i != 6)`	This evaluates to `true`	...	...
9	…	…		

As you can see from the trace table, since the value 6 is never assigned to variable `i`, this code fragment will iterate for an infinite number of times! Obviously, this code does not satisfy the property of finiteness.

Exercise 25.1-4 Counting the Total Number of Iterations

How many iterations does this code fragment perform?

```
int i;
i = -10;
while (i > 0) {
   System.out.println(i);
   i--;
}
System.out.println("The end");
```

Solution

Initially, the value −10 is assigned to variable `i`. The Boolean expression directly evaluates to `false` and the flow of execution goes right to the `System.out.println("The end")` statement. Thus, this code fragment performs zero iterations.

Exercise 25.1-5 Finding the Sum of Four Numbers

Using a pre-test loop structure, write a Java program that lets the user enter four numbers and then calculates and displays their sum.

Solution

Do you remember the example in Section 24.2 for calculating the sum of four numbers? At the end, after a little work, the proposed code fragment became

```
total = 0;
```

```
execute_these_statements_4_times {
   x = Double.parseDouble(cin.nextLine());
   total = total + x;
}
System.out.println(total);
```

Now, you need a way to "present" the statement *execute_these_statements_4_times* with real Java statements. The `while` statement can achieve this, but you need an additional variable to count the total number of iterations. This way, when the desired number of iterations has been performed, the flow of execution will exit the loop.

Following is a general purpose code fragment that iterates for the number of times specified by *total_number_of_iterations*,

```
i = 1;
while ( i <= total_number_of_iterations ) {

   A statement or block of statements

   i++;
}
```

where *total_number_of_iterations* can be a constant value or even a variable or an expression.

After combining this code fragment with the previous one, the final program becomes

Project_25.1-5

```
public static void main(String[] args) throws Exception {
   double total, x;
   int i;
   total = 0;
   i = 1;
   while (i <= 4) {
      x = Double.parseDouble(cin.nextLine()); [More…]
      total = total + x;

      i++;
   }
   System.out.println(total);
}
```

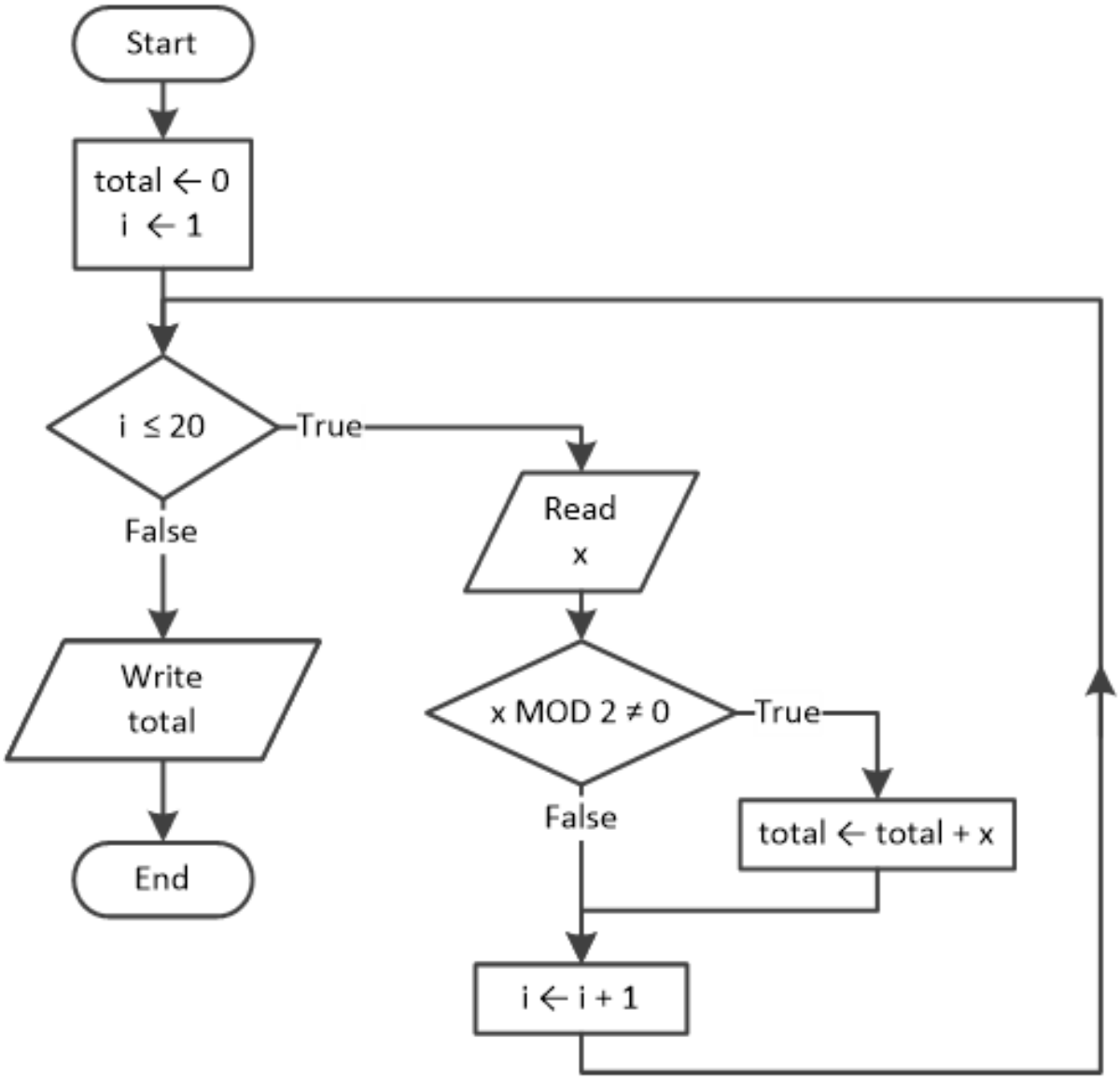
The name of the variable `i` *is not binding. You can use any variable name you wish such as* `counter`, `count`, `k`, *and more.*

Exercise 25.1-6 Finding the Sum of Odd Numbers

Design a flowchart and write the corresponding Java program that lets the user enter 20 integers, and then calculates and displays the sum of the odd numbers.

Solution

This is quite easy. What the program must do inside the loop is check whether or not a user-provided number is odd and, if it is, that number must accumulate in variable `total`; even numbers must be ignored. The flowchart is as follows. It includes a single-alternative decision structure nested within a pre-test loop structure.



The corresponding Java program is as follows.

Project_25.1-6

```
public static void main(String[] args) throws Exception {
   int total, i, x;
   total = 0;
   i = 1;
   while (i <= 20) {
      x = Integer.parseInt(cin.nextLine());
      if (x % 2 != 0) {
         total += x; //This is equivalent to total = total + x
      }
      i++;
   }
   System.out.println(total);
}
```

> ✎ *You can nest **any** decision control structure inside **any** loop control structure as long as you keep them syntactically and logically correct.*

Exercise 25.1-7 Finding the Sum of N Numbers

Write a Java program that lets the user enter N numbers and then calculates and displays their sum. The value of N must be provided by the user at the beginning of the program.

Solution

In this exercise, the total number of iterations depends on a value that the user must enter. Following is a general purpose code fragment that iterates for N times, where N is provided by the user.

```
n = Integer.parseInt(cin.nextLine());
i = 1;
while (i <= n) {
   A statement or block of statements
   i++;
}
```

According to what you have learned so far, the final program becomes

Project_25.1-7

```
public static void main(String[] args) throws Exception {
   int n, i;
```

```
   double x, total;
   total = 0;
   n = Integer.parseInt(cin.nextLine());
   i = 1;
   while (i <= n) {
      x = Double.parseDouble(cin.nextLine());
      total += x;
      i++;
   }
   System.out.println(total);
}
```

Exercise 25.1-8 Finding the Sum of an Unknown Quantity of Numbers

Write a Java program that lets the user enter integer values repeatedly until the value −1 is entered. When data input is completed, the sum of the numbers entered must be displayed. (The value of −1 must not be included in the final sum). Next, create a trace table to check if your program operates properly using 10, 20, 5, and −1 as input values.

Solution

In this exercise, the total number of iterations is unknown. If you were to use decision control structures, your program would look something like the code fragment that follows.

```
total = 0;
x = Integer.parseInt(cin.nextLine());
if (x != -1) { //Check variable x [More…]
   total += x; //and execute this statement
   x = Integer.parseInt(cin.nextLine()); //and this one
   if (x != -1) { //Check variable x
      total += x; //and execute this statement
      x = Integer.parseInt(cin.nextLine()); //and this one
      if (x != -1) { //Check variable x
         total += x; //and execute this statement
         x = Integer.parseInt(cin.nextLine()); //and this one
         …
         …
      }
   }
}
System.out.println(total);
```

Now let's rewrite this program using a loop control structure instead. The final program is presented next. If you try to follow the flow of execution, you will find that it operates equivalently to the previous one.

Project_25.1-8

```
public static void main(String[] args) throws Exception {
   double total, x;
   total = 0;
   x = Integer.parseInt(cin.nextLine());
   while (x != -1) { //Check variable x
      total += x; //and execute this statement
      x = Integer.parseInt(cin.nextLine()); //and this one
   }
   System.out.println(total);
}
```

Now let's create a trace table to determine if this program operates properly using 10, 20, 5, and −1 as input values.

Step	Statement	Notes	x	total
1	`total = 0`		?	**0**
2	`x = Integer.parseI…`		**10**	0
3	`while (x != -1)`	This evaluates to `true`		
4	`total += x`		10	**10**
5	`x = Integer.parseI…`		**20**	10
6	`while (x != -1)`	This evaluates to `true`		
7	`total += x`		20	**30**
8	`x = Integer.parseI…`		**5**	30
9	`while (x != -1)`	This evaluates to `true`		
10	`total += x`		5	**35**
11	`x = Integer.parseI…`		**−1**	35
12	`while (x != -1)`	This evaluates to `false`		
13	`.println(total)`	It displays: 35		

As you can see, in the end, variable `total` contains the value 35, which is, indeed, the sum of the values 10 + 20 + 5. Moreover, the final user-provided value of −1 does not participate in the final sum.

✎ *When the number of iterations is known before the loop starts iterating the loop is often called "definite loop". In this exercise, however, the number of iterations is not known before the loop starts iterating, and it depends on a certain condition. This type of loop is often called "indefinite loop".*

Exercise 25.1-9 Finding the Product of 20 Numbers

Write a Java program that lets the user enter 20 numbers and then calculates and displays their product.

Solution

If you were to use a sequence control structure, it would be something like the next code fragment.

```
p = 1;
x = Double.parseDouble(cin.nextLine()); [More…]
p = p * x;

x = Double.parseDouble(cin.nextLine());
p = p * x;
x = Double.parseDouble(cin.nextLine());
p = p * x;
…
…
x = Double.parseDouble(cin.nextLine());
p = p * x;
```

✎ *Note that variable* `p` *is initialized to 1 instead of 0. This is necessary for the statement* `p = p * x` *to operate properly; the final product would be zero otherwise.*

Using knowledge from the previous exercises, the final program becomes

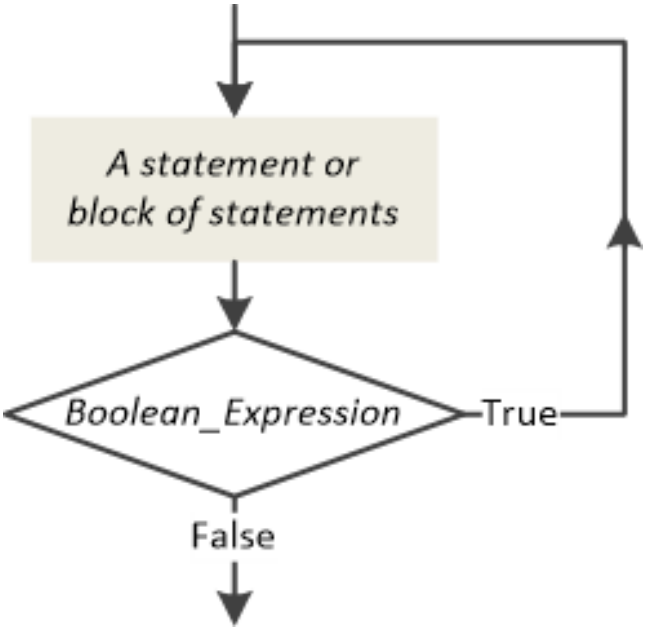
Project_25.1-9

```
public static void main(String[] args) throws Exception {
   double p, x;
   int i;
   p = 1;
```

```
    i = 1;
    while (i <= 20) {
        x = Double.parseDouble(cin.nextLine());
        p = p * x;
        i++;
    }
    System.out.println(p);
}
```

25.2 The Post-Test Loop Structure

The post-test loop structure is shown in the following flowchart.



In loop control structures, one of the diamond's exits always has an upward direction.

Let's see what happens when the flow of execution reaches a post-test loop structure. The statement or block of statements of the structure is directly executed and if *Boolean_Expression* evaluates to `true`, the flow of execution goes back to the point just above the statement or block of statements of the structure. The statement or block of statements is executed once more and if *Boolean_Expression* evaluates to `true` again, the process repeats. The iterations stop when *Boolean_Expression*, at some point, evaluates to `false` and the flow of execution exits the loop.

✎ *The post-test loop differs from the pre-test loop in that first the statement or block of statements of the structure is executed and afterwards the Boolean expression is evaluated. Consequently, the post-test loop performs at least one iteration!*

Each time the statement or block of statements of a loop control structure is executed, the term used in computer science is "the loop is iterating" or "the loop performs an iteration".

The general form of the Java statement is

```
do {
    A statement or block of statements
} while (Boolean_Expression);
```

The following example displays the numbers 1 to 10.

Project_25.2

```
public static void main(String[] args) throws Exception {
   int i;
   i = 1;
   do {
      System.out.println(i);
      i++;
   } while (i <= 10);
}
```

✎ *Note the presence of a semicolon (;) character at the end of the* `do-while` *statement.*

When only one single statement needs to be part of the `do-while` statement, you are allowed to omit the braces { }. Thus the `do-while` statement can be written as shown below.

```
do
   One_Single_Statement;
while (Boolean_Expression);
```

✎ *To prevent potential logic errors, many programmers prefer to always use braces even when the* `do-while` *statement encloses just one statement.*

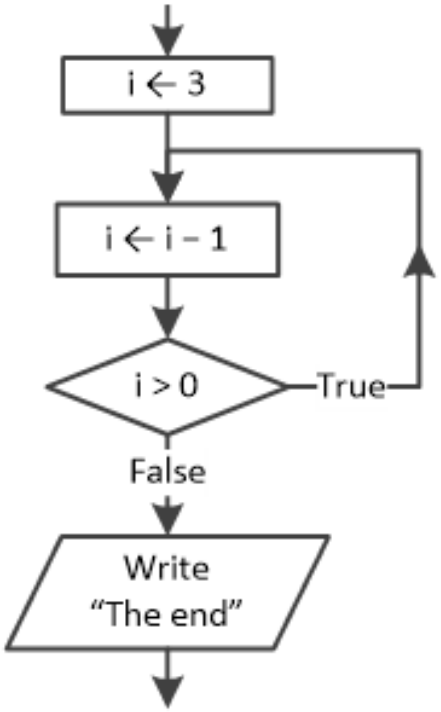
Exercise 25.2-1 Designing the Flowchart and Counting the Total Number of Iterations

Design the corresponding flowchart for the following code fragment. How many iterations does this Java code perform?

```
int i;
i = 3;
do {
   i--;
} while (i > 0);
System.out.println("The end");
```

Solution

The corresponding flowchart fragment is as follows.



Now, let's create a trace table to observe the flow of execution.

Step	Statement	Notes	i	
1	`i = 3`		3	
2	`i--`		2	1st iteration
3	`while (i > 0)`	This evaluates to `true`		
4	`i--`		1	2nd iteration
5	`while (i > 0)`	This evaluates to `true`		

6	`i--`		**0**	3rd iteration
7	`while (i > 0)`	This evaluates to `false`		
8	`.println("The end")`	It displays: The end		

As you can see from the trace table, the total number of iterations is **three**.

> ✎ *Both the statement or block of statements of a post-test loop structure is executed N times, and the Boolean expression is evaluated N times. Therefore, to determine the total number of iterations, you can count either the number of times the statement or block of statements is executed, or the number of times the Boolean expression is evaluated. Both counts are equal!*

Exercise 25.2-2 Counting the Total Number of Iterations

How many iterations does this code fragment perform?

```
int i;
i = 3;
do {
   System.out.println(i);
   i--;
} while (i >= 0);
System.out.println("The end");
```

Solution

This exercise is almost identical to the previous one. The main difference is that the Boolean expression here remains `true`, even for `i = 0`. Therefore, it performs an additional iteration, that is, **four** iterations.

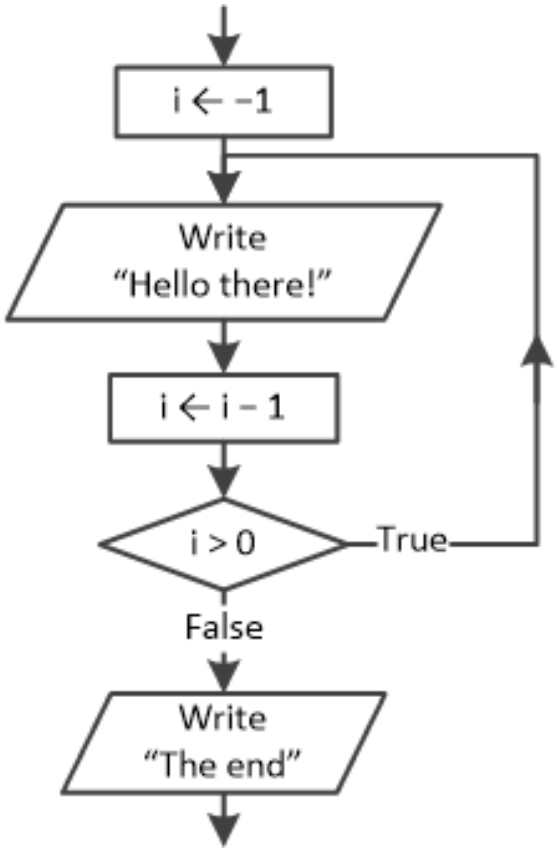
Exercise 25.2-3 Designing the Flowchart and Counting the Total Number of Iterations

Design the corresponding flowchart for the following code fragment. How many iterations does this code perform?

```
int i;
i = -1;
do {
   System.out.println("Hello there!");
   i--;
} while (i > 0);
System.out.println("The end");
```

Solution

The corresponding flowchart fragment is as follows.



Initially the value −1 is assigned to the variable `i`. Inside the loop, the message “Hello there!” is displayed and variable `i` is decremented by one (resulting in the value −2). The Boolean expression `i > 0` evaluates to `false`, and the flow of execution proceeds directly to the `Write ("The end")` statement. Thus, this algorithm performs one iteration!

Exercise 25.2-4 Counting the Total Number of Iterations

How many iterations does this code fragment perform?

```
int i;
i = 1;
do {
   i = i + 2;
} while (i != 4);
System.out.println("The end");
```

Solution

Let's create a trace table to observe the flow of execution.

Step	Statement	Notes	i		
1	`i = 1`		1		
2	`i = i + 2`		3	1st	iteration
3	`while (i != 4)`	This evaluates to `true`			
4	`i = i + 2`		5	2nd	iteration
5	`while (i != 4)`	This evaluates to `true`			
6	`i = i + 2`		7	3rd	iteration
7	`while (i != 4)`	This evaluates to `true`			
8	…	…		…	…
9	…	…			

As you can see from the trace table, since the value 4 is never assigned to variable `i`, this code fragment will iterate for an infinite number of times! Obviously, this code does not satisfy the property of finiteness.

Exercise 25.2-5 Finding the Product of N Numbers

Write a Java program that lets the user enter N numbers and then calculates and displays their product. The value of N must be provided by the user at the beginning of the program. What happens if you switch the post-test loop structure with a pre-test loop structure? Do both programs operate exactly the same way for all possible input values of N?

Solution

Both programs below let the user enter N numbers, calculate, and display their product. The left one uses a pre-test, while the right one uses a post-test loop structure. If you try to execute them and enter any value greater than zero for N, both programs operate exactly the same way!

Project_25.2-5a

```
public static void main(String[] args)
                    throws Exception {
   int n, i;
   double p, x;
   n = Integer.parseInt(cin.nextLine());
```

```
   p = 1;
   i = 1;
   while (i <= n) {
      x = Double.parseDouble(cin.nextLine());
      p = p * x;
      i++;
   }
   System.out.println(p);
}
```

Project_25.2-5b

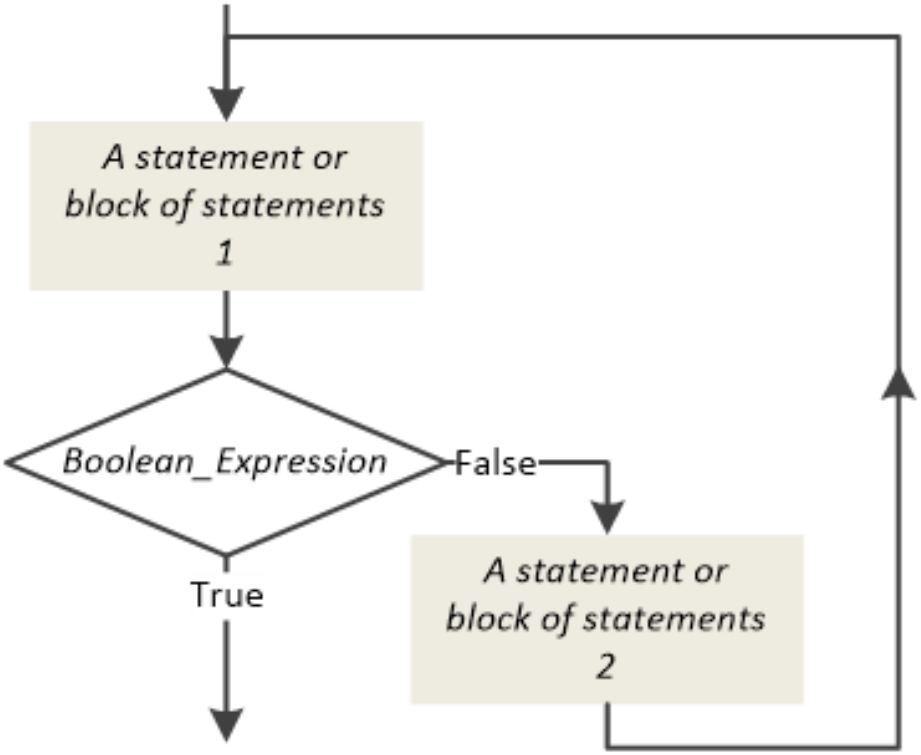
```
public static void main(String[] args)
                  throws Exception {
   int n, i;
   double p, x;
   n = Integer.parseInt(cin.nextLine());
   p = 1;
   i = 1;
   do {
      x = Double.parseDouble(cin.nextLine());
      p = p * x;
      i++;
   } while (i <= n);
   System.out.println(p);
}
```

The two Java programs, however, operate in different ways when the user enters a non-positive[18] value for N. For example, if the value 0 is entered, the left program performs **zero** iterations whereas the right program performs **one** iteration. Obviously, the left program is the right choice to solve this exercise!

✎ *A pre-test loop structure may perform zero iterations in contrast to the post-test loop structure, which performs **at least** one iteration!*

25.3 The Mid-Test Loop Structure

The mid-test loop structure is shown in the following flowchart.



Let's see what happens when the flow of execution reaches a mid-test loop structure. The statement or block of statements 1 of the structure is directly executed and if *Boolean_Expression* evaluates to `false`, the statement or block of statements 2 is executed and the flow of execution goes back to the point just above the statement or block of statements 1 of the structure. The statement or block of statements 1 is executed once more and if *Boolean_Expression* evaluates to `false` again, the process repeats. The iterations stop when *Boolean_Expression*, at some point, evaluates to `true` and the flow of execution exits the loop.

Although this loop control structure is directly supported in some computer languages such as Ada, unfortunately this is not true for Java. However, you can still write mid-test loops using the `while` statement along with an `if` and a `break` statement. The main idea is to create an endless loop and break out of it when the Boolean expression that exists between the two statements (or block of statements) of the structure evaluates to `true`. The idea is shown in the code fragment given in general form that follows.

```
while (true) {

    A statement or block of statements 1

    if (Boolean_Expression) break;
```

```
    A statement or block of statements 2
}
```

> ✎ *You can break out of a loop before it actually completes all of its iterations by using the* `break` *statement.*

The following example displays the numbers 1 to 10.

Project_25.3

```
public static void main(String[] args) throws Exception {
   int i;
   i = 1;
   while (true) {
      System.out.println(i);
      if (i >= 10) break;
      i++;
   }
}
```

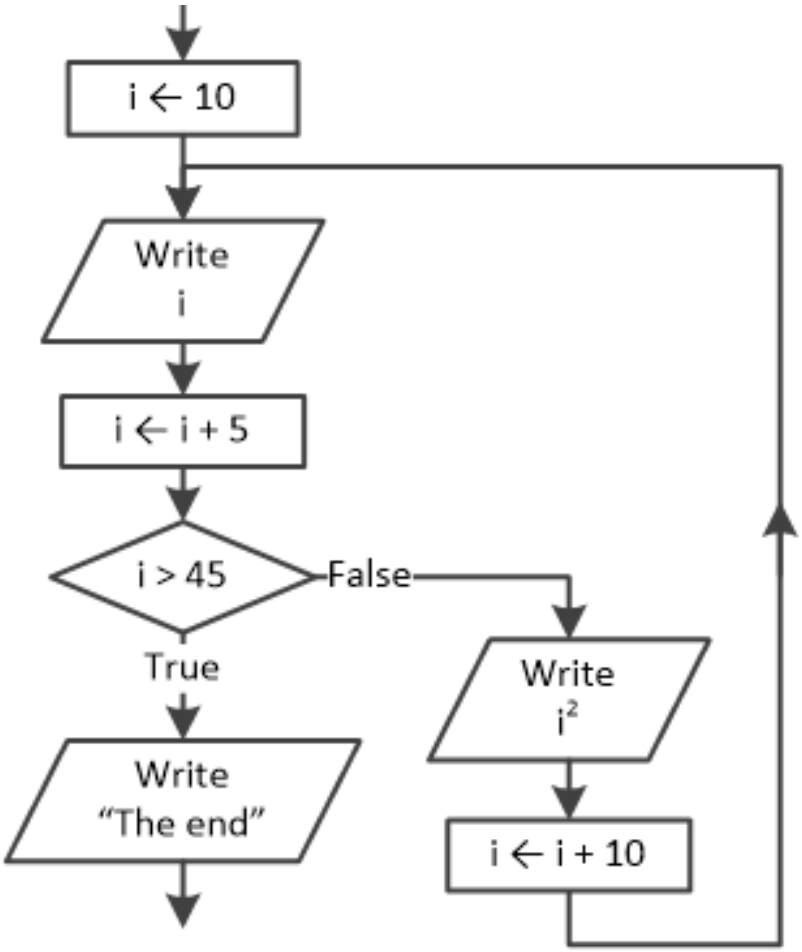
Exercise 25.3-1 Designing the Flowchart and Counting the Total Number of Iterations

Design the corresponding flowchart for the following code fragment and create a trace table to determine the values of variable `i` *in each step.*

```
int i;
i = 10;
while (true) {
   System.out.println(i);
   i += 5;
   if (i > 45) break;
   System.out.println(i * i);
   i += 10;
}
System.out.println("The end");
```

Solution

The corresponding flowchart fragment is as follows.



Now, let's create a trace table to observe the flow of execution.

Step	Statement	Notes	i
1	`i = 10`		**10**
2	`.println(i)`	It displays: 10	
3	`i += 5`		**15**
4	`if (i > 45)`	This evaluates to `false`	
5	`.println(i * i)`	It displays: 225	
6	`i += 10`		**25**
7	`.println(i)`	It displays: 25	
8	`i += 5`		**30**
9	`if (i > 45)`	This evaluates to `false`	

10	`.println(i * i)`	It displays: 900	
11	`i += 10`		**40**
12	`.println(i)`	It displays: 40	
13	`i += 5`		**45**
14	`if (i > 45)`	This evaluates to `false`	
15	`.println(i * i)`	It displays: 2025	
16	`i += 10`		**55**
17	`.println(i)`	It displays: 55	
18	`i += 5`		**60**
19	`if (i > 45)`	This evaluates to `true`	
20	`.println("The end")`	It displays: The end	

25.4 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A pre-test loop may perform zero iterations.
2) In flowcharts, both exits of the diamond symbol in a pre-test loop structure, have an upwards direction.
3) The statement or block of statements of a pre-test loop structure is executed at least one time.
4) A `while` statement stops iterating when its Boolean expression evaluates to `true`
5) In a pre-test loop structure, when the statement or block of statements of the structure is executed N times, the Boolean expression is evaluated N − 1 times.
6) A post-test loop may perform zero iterations.
7) In a post-test loop structure, when the statement or block of statements of the structure is executed N times, its Boolean expression is evaluated N times as well.

8) You cannot nest a decision control structure inside a post-test loop structure.

9) In the mid-test loop structure, the statement or block of statements 1 is executed the same number of times as the statement or block of statements 2.

10) In the following code fragment, the word “Hello” is displayed 10 times.

```
int i = 1;
while (i <= 10)
   System.out.println("Hello");
   i++;
```

11) The following Java program does **not** satisfy the property of finiteness.

```
public static void main(String[] args) throws Exception {
   int i;
   i = 1;
   while (i != 10) {
      System.out.println("Hello");
      i += 2;
   }
}
```

12) In the following code fragment, the word “Hello” is displayed an infinite number of times.

```
int i = 1;
do {
   System.out.println("Hello");
} while (i >= 10);
```

13) The following Java program satisfies the property of effectiveness.

```
public static void main(String[] args) throws Exception {
   int i;
   do {
      System.out.println("Hello");
      i -= 2;
   } while (i > 10);
}
```

14) The following code fragment does **not** satisfy the property of definiteness.

```
int b;
double a;
```

```
b = Integer.parseInt(cin.nextLine());
if (b != 1) {
   do {
      a = 1 / (b − 1);
      b++;
   } while (b <= 10);
}
```

15) In the following code fragment, the word “Zeus” is displayed 10 times.

```
int i = 1;
while (true) {
   System.out.println("Zeus");
   if (i > 10) break;
   i++;
}
```

25.5 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) In flowcharts, the diamond symbol is being used
   a) in decision control structures.
   b) in loop control structures.
   c) all of the above
2) A post-test loop structure
   a) performs one iteration more than the pre-test loop structure does.
   b) performs the same number of iterations as the pre-test loop structure does.
   c) it depends
3) In a post-test loop structure, the statement or block of statements of the structure
   a) are executed before the loop's Boolean expression is evaluated.
   b) are executed after the loop's Boolean expression is evaluated.
   c) none of the above
4) In the following code fragment

```
int i = 1;
while (i < 10) {
   System.out.println("Hello Hermes");
```

```
   i++;
}
```

the message “Hello Hermes” is displayed

a) 10 times.
b) 9 times.
c) 1 time.
d) 0 times.
e) none of the above

5) In the following code fragment

```
int i = 1;
while (i < 10)
   System.out.println("Hi!");
   System.out.println("Hello Ares");
   i++;
```

the message “Hello Ares” is displayed

a) 10 times.
b) 11 time.
c) 1 times.
d) 0 times.
e) none of the above

6) In the following code fragment

```
int i = 1;
while (i < 10)
   i++;
   System.out.println("Hi!");
   System.out.println("Hello Aphrodite");
```

the message “Hello Aphrodite” is displayed

a) 10 times.
b) 1 time.
c) 0 times.
d) none of the above

7) In the following code fragment

```
int i = 1;
while (i >= 10) {
```

```
   System.out.println("Hi!");
   System.out.println("Hello Apollo");
   i++;
}
```

the message “Hello Apollo” is displayed

a) 10 times.
b) 1 time.
c) 0 times.
d) none of the above

8) The following code fragment

```
int i, n;
double s;
n = Integer.parseInt(cin.nextLine());
s = 0;
i = 1;
while (i < n) {
   a = Double.parseDouble(cin.nextLine());
   s = s + a;
   i++;
}
System.out.println(s);
```

calculates and displays the sum of

a) as many numbers as the value of variable `n` denotes.
b) as many numbers as the result of the expression `n − 1` denotes.
c) as many numbers as the value of variable `i` denotes.
d) none of the above

9) In the following code fragment

```
int i = 1;
do {
   System.out.println("Hello Poseidon");
   i++;
} while (i > 5);
```

the message “Hello Poseidon” is displayed

a) 5 times.
b) 1 time.
c) 0 times.

d) none of the above

10) In the following code fragment

```
int i = 1;
do {
   System.out.println("Hello Athena");
   i += 5;
} while (i != 50);
```

the message “Hello Athena” is displayed

a) at least one time.
b) at least 10 times.
c) an infinite number of times.
d) all of the above

11) In the following code fragment

```
int i = 0;
do {
   System.out.println("Hello Apollo");
} while (i > 10);
```

the message “Hello Apollo” is displayed

a) at least one time.
b) an infinite number of times.
c) none of the above

12) In the following code fragment

```
int i = 10;
while (true) {
   i--;
   if (i > 0) break;
   System.out.println("Hello Aphrodite");
}
```

the message “Hello Aphrodite” is displayed

a) at least one time.
b) an infinite number of times.
c) ten times
d) none of the above

25.6 Review Exercises

Complete the following exercises.

1) Identify the error(s) in the following Java program. It must display the numbers 3, 2, 1 and the message “The end”.

```
public static void main(String[] args) throws Exception {
   int i;
   i = 3;
   do
      System.out.println(i);
      i--;
   } while (i >= 0)
   System.out.println(The end);
}
```

2) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this Java program perform?

```
int i, x;
i = 3;
x = 0;
while (i >= 0) {
   i--;
   x += i;
}
System.out.println(x);
```

3) Design the corresponding flowchart and create a trace table to determine the values of the variables in each step of the next Java program. How many iterations does this Java program perform?

```
public static void main(String[] args) throws Exception {
   int i;
   i = -5;
   while (i < 10) {
      i--;
   }
   System.out.println(i);
}
```

4) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this Java program perform?

```
int a, b, c, d;
a = 2;
while (a <= 10) {
```

```
   b = a + 1;
   c = b * 2;
   d = c - b + 1;
   switch (d) {
      case 4:
         System.out.println(b + ", " + c);
         break;
      case 5:
         System.out.println(c);
         break;
      case 8:
         System.out.println(a + ", " + b);
         break;
      default:
         System.out.println(a + ", " + b + ", " + d);
   }
   a += 4;
}
```

5) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this Java code perform?

```
int a, b, c, d, x;
a = 1;
b = 1;
c = 0;
d = 0;
while (b < 2) {
   x = a + b;
   if (x % 2 != 0)
      c = c + 1;
   else
      d = d + 1;
   a = b;
   b = c;
   c = d;
}
```

6) Fill in the gaps in the following code fragments so that all loops perform exactly four iterations.

i)

```
int a = 3;
while (a > …… ) {
   System.out.println(a);
```

```
    a--;
}
```

ii)

```
int a = 5;
while (a < …… ) {
    System.out.println(a);
    a++;
}
```

iii)

```
double a = 9;
while (a != 10 ) {
    System.out.println(a);
    a = a + …… ;
}
```

iv)

```
int a = 1;
while (a != …… ) {
    System.out.println(a);
    a -= 2;
}
```

v)

```
int a = 2;
while (a < …… ) {
    System.out.println(a);
    a = 2 * a;
}
```

vi)

```
double a = 1;
while (a < …… ) {
    System.out.println(a);
    a = a + 0.1;
}
```

7) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this code perform?

```
int y, x;
y = 5;
x = 38;
do {
    y *= 2;
```

```
   x++;
   System.out.println(y);
} while (y < x);
```

8) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this code perform?

```
int x;
x = 1;
do {
   if (x % 2 == 0) {
      x++;
   }
   else {
      x += 3;
   }
   System.out.println(x);
} while (x < 12);
```

9) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this code perform?

```
double x, y;
y = 2;
x = 0;
do {
   y = Math.pow(y, 2);
   if (x < 256) {
      x = x + y;
   }
   System.out.println(x + ", " + y);
} while (y < 65535);
```

10) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this code perform?

```
int a, b, c, d, x;
a = 2;
b = 4;
c = 0;
d = 0;
do {
   x = a + b;
   if (x % 2 != 0) {
```

```
      c = c + 5;
   }
   else if (d % 2 == 0) {
      d = d + 5;
   }
   else {
      c = c + 3;
   }
   a = b;
   b = d;
} while (c < 11);
```

11) Fill in the gaps in the following code fragments so that all loops perform exactly six iterations.

i)

```
int a = 5;
do {
   System.out.println(a);
   a--;
} while (a > …… );
```

ii)

```
int a = 12;
do {
   System.out.println(a);
   a++;
} while (a < …… );
```

iii)

```
double a = 20;
do {
   System.out.println(a);
   a = a + …… ;
} while (a != 23);
```

iv)

```
int a = 100;
do {
   System.out.println(a);
   a -= 20;
} while (a != …… );
```

v)

```
int a = 2;
do {
```

```
   System.out.println(a);
   a = 2 * a;
} while (a != …… );
```

vi)

```
double a = 10;
do {
   System.out.println(a);
   a = a + 0.25;
} while (a <= …… );
```

12) Fill in the gaps in the following code fragments so that all display the value 10 (or 10.0) at the end.

i)

```
int x = 0;
int y = 0;
do {
   x++;
   y += 2;
} while (x <= …… );
System.out.println(y);
```

ii)

```
int x = 1;
double y = 20;
do {
   x--;
   y -= 2.5;
} while (x >= …… );
System.out.println(y);
```

iii)

```
int x = 3;
double y = 2.5;
do {
   x--;
   y *= 2;
} while (x >= …… );
System.out.println(y);
```

iv)

```
int x = 30;
int y = 101532;
do {
   x -= ……;
```

```
   y = (int)(y / 10);
} while (x >= 0);
System.out.println(y);
```

13) Using a pre-test loop structure, write a Java program that lets the user enter N numbers and then calculates and displays their sum and their average. The value of N must be provided by the user at the beginning of the program.

14) Using a pre-test loop structure, write a Java program that lets the user enter N integers and then calculates and displays the product of those that are even. The value of N must be provided by the user at the beginning of the program. Moreover, if all user-provided integers are odd, the message “You entered no even integers” must be displayed.

15) Using a pre-test loop structure, write a Java program that lets the user enter 100 integers and then calculates and displays the sum of those with a last digit of 0. For example, the values 10, 2130, and 500 are such numbers.

    Hint: You can isolate the last digit of any integer using a modulus 10 operation.

16) Using a pre-test loop structure, write a Java program that lets the user enter 20 integers and then calculates and displays the sum of those that consist of three digits.

    Hint: All three-digit integers are between 100 and 999.

17) Using a pre-test loop structure, write a Java program that lets the user enter numeric values repeatedly until the value 0 is entered. When data input is completed, the product of the numbers entered must be displayed. (The last 0 entered must not be included in the final product). Next, create a trace table to check if your program operates properly using 3, 2, 9, and 0 as input values.

18) The population of a town is now at 30000 and is expanding at a rate of 3% per year. Using a pre-test loop structure, write a Java program to determine how many years it will take for the population to exceed 100000.

19) Using a post-test loop structure, design a flowchart and write the corresponding Java program that lets the user enter 50 integers and

then calculates and displays the sum of those that are odd and the sum of those that are even.

20) Using a post-test loop structure, write a Java program that lets the user enter N integers and then calculates and displays the product of those that are negative. The value of N must be provided by the user at the beginning of the program, and the final product must always be displayed as a positive value. Assume that the user enters a value greater than 0 for N.

21) Using a post-test loop structure, write a Java program that prompts the user to enter five integers and then calculates and displays the product of all three-digit integers with a first digit of 5. For example, the values 512, 555, and 593 are all such numbers

    Hint: All three-digit integers with a first digit of 5 are between 500 and 599.

22) The current population of a beehive is 50,000. Each year, the beehive experiences a 5% increase due to new births, but also faces a 15% mortality rate due to environmental reasons. Using a post-test loop structure, write a Java program to determine how many years it will take for the population to fall below 20,000.

Chapter 26
Definite Loops

26.1 The `for` statement

In Chapter 25, as you certainly noticed, the `while` statement was used to iterate for both a known number and an unknown number of times (in situations where the number of iterations was not known at the time the loop started iterating). In other words, the `while` statement was used to create both definite and indefinite loops.

Since definite loops are so frequently used in computer programming, almost every computer language, including Java, incorporates a special statement that is notably more readable and convenient than the `while` statement—and this is the `for` statement.

The general form of the `for` statement, is

for (*initialize_counter*; *evaluate_counter; update_counter*) {

```
    A statement or block of statements
```

}

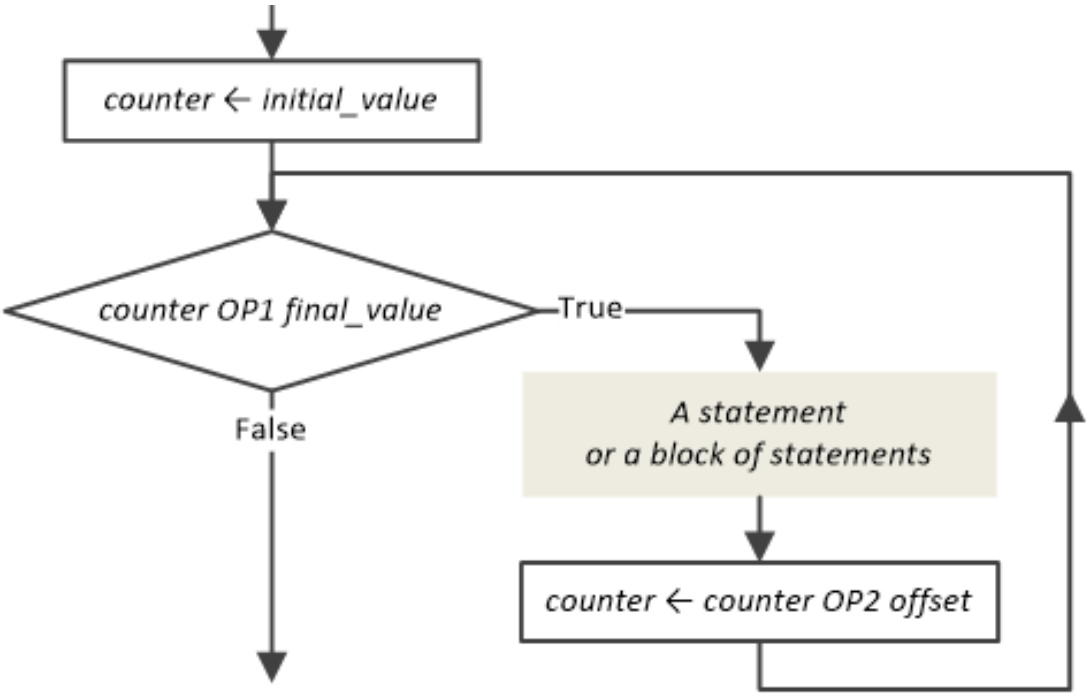
where

- *`counter`* is a variable.
- *`initialize_counter`* must be a statement that assigns an initial value to the variable *`counter`*.
- *`evaluate_counter`* must be a Boolean expression that evaluates the variable *`counter`*.
- *`update_counter`* must be a statement that alters (usually increments or decrements) the value of variable *`counter`*.

Even though the `for` statement can be written in many different ways in Java, this book deals only with its basic form. So, when a `for` statement is used to create define loops, its general form in a more convenient format can be as follows.

```
for (counter = initial_value; counter OP1 final_value; counter = counter OP2 offset) {
    A statement or block of statements
}
```

And the corresponding flowchart is shown here.



where

- *initial_value*, *final_value* and *offset* can be constant values, variables or expressions. Negative values are also permitted.
- operator *OP1* should be <= when *counter* increments and >= when *counter* decrements.
- operator *OP2* should be + when *counter* increments and – when *counter* decrements.

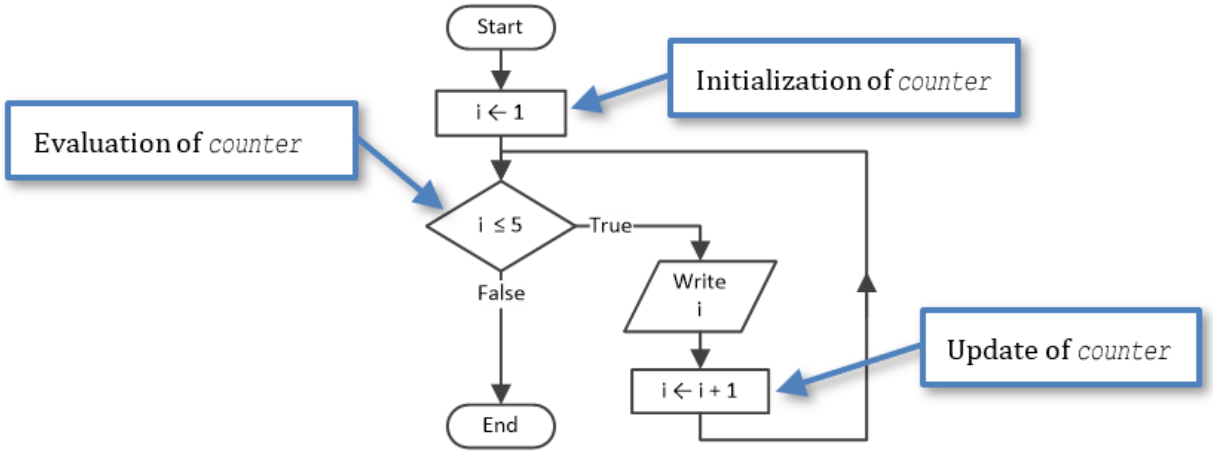
> ✎ *A for-loop is actually a pre-test loop structure.*

The following example displays the numbers 1, 2, 3, 4, and 5. Variable *counter* (here `i`) increments from 1 to 6, allowing the loop to perform 5 iterations.

```
Project_26.1a
public static void main(String[] args) throws Exception {
                int i;
        for (i = 1; i <= 5; i = i + 1) {
                System.out.println(i);
                }
        }
```

✎ *When the for-loop performs its last iteration, it displays the content of variable* `i`*, which is 5, then variable* `i` *increments by one (it becomes 6), and the flow of execution exits the loop.*

The flowchart for the program that you have just seen is presented below. When the flow of execution reaches the for-loop, the initial value 1 is assigned to variable `i`, the Boolean expression `i ≤ 5` evaluates to `true`, the statement `Write i` is executed and the value 1 is displayed on the screen. The variable `i` increments by one and the flow of execution goes back to the point just above the diamond symbol. The Boolean expression evaluates to `true` again the value 2 is displayed and the process repeats. In the 5th iteration, the value 5 is displayed, the value of variable `i` becomes 6, the Boolean expression evaluates to `false` and the flow of execution exits the loop.



When variable *counter* increments (or decrements) by 1, it is more convenient to use compound assignment operators or even incrementing (or decrementing) operators. The next example displays the numbers from 1 to 10 using the incrementing operator ++.

```
Project_26.1b
public static void main(String[] args) throws Exception {
                        int i;
                for (i = 1; i <= 10; i++) {
                    System.out.println(i);
                            }
                        }
```

The following example displays even numbers from 10 to 2 using the compound assignment operator `-=`.

```
                    Project_26.1c
public static void main(String[] args) throws Exception {
                            int i;
                for (i = 10; i >= 2; i -= 2) {
                     System.out.println(i);
                              }
                             }
```

> *Note that the comparison operator should be* `>=` *when* `counter` *(here* `i`*) decrements and* `<=` *when* `counter` *increments.*

Although it is not recommended, since a for-loop is actually a pre-test loop structure, you can replace the `for` statement with a `while` statement. The previous example can be written using a `while` statement as follows.

```
                    Project_26.1d
public static void main(String[] args) throws Exception {
                            int i;
                           i = 10;
                      while (i >= 2) {
                     System.out.println(i);
                            i -= 2;
                              }
                             }
```

The following example displays even numbers from −2 to −10 using variables instead of constant values for *`initial_value`*, *`final_value`*, and *`offset`*.

```
                    Project_26.1e
public static void main(String[] args) throws Exception {
       int i; int x1 = -2; int x2 = -10; int t = -2;
                for (i = x1; i >= x2; i += t) {
                     System.out.println(i);
                              }
                             }
```

Don't ever dare alter the value of `counter` (here `i`) inside the loop! The same applies to `initial_value` (here `x1`), `final_value` (here `x2`), and `offset` (here `t`). This makes your code unreadable and could lead to incorrect results. If you insist, though, please use a `while` statement instead.

The following example displays the letters “H”, “e”, “l”, “l”, and “o” (all without the double quotes).

Project_26.1f

```
public static void main(String[] args) throws Exception {
            int i; String message = "Hello";
       for (i = 0; i <= message.length() - 1; i++) {
            System.out.println(message.charAt(i));
                          }
                        }
```

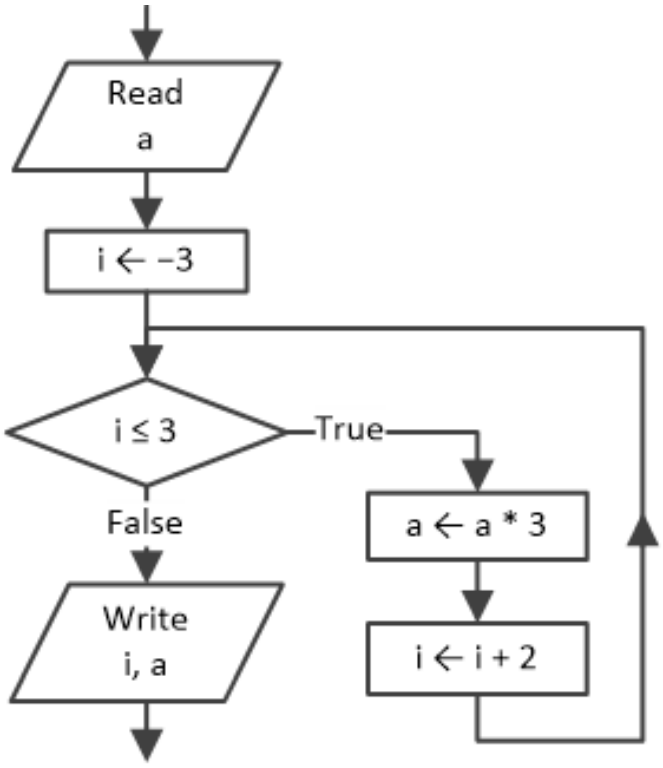
The `length()` method returns the number of characters variable `message` consists of, whereas the `message.charAt(i)` method returns the character located at the specified position (indicated by the variable `i`) in the string variable `message` (see Section 14.3).

Exercise 26.1-1 Creating the Trace Table

Design the corresponding flowchart and create a trace table to determine the values of the variables in each step of the next code fragment when the input value 1 is entered.

```
int a, i;
a = Integer.parseInt(cin.nextLine());
for (i = -3; i <= 3; i += 2) {
   a = a * 3;
}
System.out.println(i + " " + a);
```

Solution The corresponding flowchart fragment is as follows.



You should always keep in mind that a for-loop is actually a pre-test loop structure!

If you rewrite the code fragment using the `while` statement the result is as follows.

```
int a, i;
a = Integer.parseInt(cin.nextLine());
i = -3;
while (i <= 3) {
   a = a * 3;
   i += 2;
}
System.out.println(i + " " + a);
```

Now, in order to create a trace table for a `for` statement you have two choices: you can use either the corresponding flowchart or the equivalent program written with the `while` statement.

Step	Statement	Notes	a	i
1	`a = Integer.parseI…`		**1**	?
2	`i = -3`		1	**−3**

3	`i <= 3`	This evaluates to `true`			1st iteration
4	`a = a * 3`		**3**	−3	
5	`i += 2`		3	**−1**	
6	`i <= 3`	This evaluates to `true`			
7	`a = a * 3`		**9**	−1	2nd iteration
8	`i += 2`		9	**1**	
9	`i <= 3`	This evaluates to `true`			
10	`a = a * 3`		**27**	1	3rd iteration
11	`i += 2`		27	**3**	
12	`i <= 3`	This evaluates to `true`			
13	`a = a * 3`		**81**	3	4th iteration
14	`i += 2`		81	**5**	
15	`i <= 3`	This evaluates to `false`			
16	`.println(i + " " + a)`	It displays: 5 81			

✎ *Note that the flow of execution exits the loop when the value of counter `i` exceeds `final_value`. In this example, when the flow of execution does finally exit the loop, counter `i` does not contain the final value 3 but the next one in order, which is the value 5.*

Exercise 26.1-2 Creating the Trace Table

Create a trace table to determine the values of the variables in each step of the next code fragment when the input value 4 is entered.

```
int a, i; a = Integer.parseInt(cin.nextLine());
for (i = 6; i >= a; i--) {
   System.out.println(i); }
```

Solution As in the previous exercise, to create the trace table, the code should be rewritten using the `while` statement.

```
int a, i; a = Integer.parseInt(cin.nextLine());
i = 6;
while (i >= a) {
```

```
   System.out.println(i); i--;
}
```

Following is the trace table used to determine the values of the variables in each step.

Step	Statement	Notes	a	i
1	`a = Integer.parseI…`		**4**	?
2	`i = 6`		4	**6**
3	`i >= a`	This evaluates to `true`		
4	`.println(i)`	It displays: 6		
5	`i--`		4	**5**
6	`i >= a`	This evaluates to `true`		
7	`.println(i)`	It displays: 5		
8	`i--`		4	**4**
9	`i >= a`	This evaluates to `true`		
10	`.println(i)`	It displays: 4		
11	`i--`		4	**3**
12	`i >= a`	This evaluates to `false`		

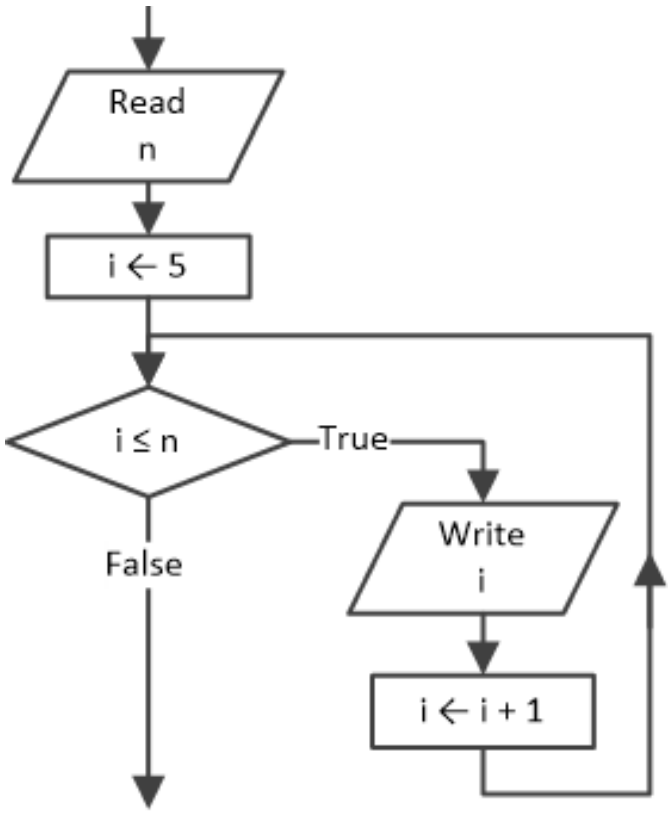
Exercise 26.1-3 Counting the Total Number of Iterations

Count the total number of iterations performed by the following code fragment for two different executions.

The input values for the two executions are: (i) 6, and (ii) 5.

```
n = Integer.parseInt(cin.nextLine()); for (i = 5; i <= n; i++) {
   System.out.println(i); }
```

Solution In order to better understand what really goes on, instead of creating a trace table, you can just design its corresponding flowchart fragment.



From this flowchart fragment you can see that i) for the input value 5, the Boolean expression evaluates to `true` and the flow of execution enters the loop. Variable `i` increases to 6, the Boolean expression evaluates to `false`, and the flow of execution exits the loop. Thus, the loop performs one iteration.

ii) for the input value 6 the loop obviously performs two iterations.

Exercise 26.1-4 Finding the Sum of Four Numbers

Write a Java program that prompts the user to enter four numbers and then calculates and displays their sum.

Solution In Exercise 25.1-5, the solution proposed with a `while` statement was the following:

```
public static void main(String[] args) throws Exception {
    double total, x; int i;
    total = 0;
    i = 1;
    while (i <= 4) {
        x = Double.parseDouble(cin.nextLine());
        total = total + x;
```

```
        i++;
    }
    System.out.println(total); }
```

It's now very easy to rewrite this using a `for` statement and have it display a prompt message before every data input.

Project_26.1-4

```
public static void main(String[] args) throws Exception {
   int i; double total, x;
   total = 0;
   for (i = 1; i <= 4; i++) {
      System.out.print("Enter a number: ");
      x = Double.parseDouble(cin.nextLine());
      total += x;
   }
   System.out.println(total); }
```

Note the absence of the `i++` statement inside the loop control structure. In a `for` statement, the counter (here variable `i`) automatically updates (here increases) at the end of each loop iteration.

Exercise 26.1-5 Finding the Square Roots from 0 to N

Write a Java program that prompts the user to enter an integer and then calculates and displays the square root of all integers from 0 to that user-provided integer.

Solution This exercise is straightforward. The user enters an integer, and the program iterates as many times as indicated by that integer. The Java program is as follows.

Project_26.1-5

```
public static void main(String[] args) throws Exception {
   int num, i;
   System.out.print("Enter an integer: "); num = Integer.parseInt(cin.nextLine());
   for (i = 0; i <= num; i++) {
      System.out.println(Math.sqrt(i));
   }
}
```

Exercise 26.1-6 Finding the Sum of 1 + 2 + 3 + … + 100

Write a Java program that calculates and displays the following sum: S = 1 + 2 + 3 + … + 100

Solution If you were to use a sequence control structure to solve this exercise, it would be something like the next code fragment.

```
s = 0;
i = 1;
s = s + i;
i = i + 1;
s = s + i;
i = i + 1;
…
…
s = s + i;
i = i + 1;
```

Let's use a trace table to better understand it.

Step	Statement	Notes	i	s
1	`s = 0`	0	?	**0**
2	`i = 1`		**1**	0
3	`s = s + i`	0 + 1 = **1**	1	**1**
4	`i = i + 1`		**2**	1
5	`s = s + i`	0 + 1 + 2 = **3**	2	**3**
6	`i = i + 1`		**3**	3
7	`s = s + i`	0 +1 + 2 + 3 = **6**	3	**6**
8	`i = i + 1`		**4**	6
...	…		…	…
...	…		…	…
199	`s = s + i`		99	**4950**
200	`i = i + 1`		**100**	4950
201	`s = s + i`	0 + 1 + 2 + 3 + …+ 99 + 100 = **5050**	100	**5050**
202	`i = i + 1`		**101**	5050

Now that everything has been cleared up, you can do the same thing, this time using a for-loop that increments variable `i` by 1.

Project_26.1-6

```
public static void main(String[] args) throws Exception {
   int s, i;
   s = 0;
   for (i = 1; i <= 100; i++) {
      s = s + i;
   }
   System.out.println(s); }
```

Exercise 26.1-7 Finding the Product of 2 × 4 × 6 × 8 × 10

Write a Java program that calculates and displays the following product: P = 2 × 4 × 6 × 8 × 10

Solution Let's solve this exercise using the following sequence control structure. Variable `p` must be initialized to 1 instead of 0. This is necessary for the statement `p = p * i` to operate properly; the final product would be zero otherwise.

```
p = 1;
i = 2;
p = p * i;
i += 2;
p = p * i;
i += 2;
p = p * i;
i += 2;
p = p * i;
i += 2;
p = p * i;
i += 2;
```

As in the previous exercise (Exercise 26.1-6), this sequence control structure can be replaced by a for-loop, as follows.

Project_26.1-7

```
public static void main(String[] args) throws Exception {
   int p, i;
   p = 1;
   for (i = 2; i <= 10; i += 2) {
      p = p * i;
   }
   System.out.println(p); }
```

Exercise 26.1-8 Finding the Sum of $2^2 + 4^2 + 6^2 + \ldots (2N)^2$

Write a Java program that lets the user enter an integer N and then calculates and displays the following sum: $S = 2^2 + 4^2 + 6^2 + \ldots (2N)^2$

Solution In this exercise, variable `i` must increment by 2. In each iteration though, its value must be raised to the second power before it is accumulated in variable `s`. The final Java program is as follows.

Project_26.1-8

```
public static void main(String[] args) throws Exception {
   int N, i; double s;
   N = Integer.parseInt(cin.nextLine()); s = 0;
   for (i = 2; i <= 2 * N; i += 2) {
      s = s + Math.pow(i, 2);
   }
   System.out.println(s); }
```

Exercise 26.1-9 Finding the Sum of $3^3 + 6^6 + 9^9 + \ldots (3N)^{3N}$

Write a Java program that lets the user enter an integer N and then calculates and displays the following sum: $S = 3^3 + 6^6 + 9^9 + \ldots + (3N)^{3N}$

Solution This is pretty much the same as the previous exercise. The only difference is that variable `i` must be raised to the i^{th} power before it is accumulated in variable `s`. Using the for-loop, the final Java program is as follows.

Project_26.1-9

```
public static void main(String[] args) throws Exception {
   int N, i; double s;
   N = Integer.parseInt(cin.nextLine()); s = 0;
   for (i = 3; i <= 3 * N; i += 3) {
      s = s + Math.pow(i, i);
   }
   System.out.println(s); }
```

Exercise 26.1-10 Finding the Average Value of Positive Numbers

Write a Java program that lets the user enter 100 numbers and then calculates and displays the average value of the positive numbers. Add all necessary checks to make the program satisfy the property of definiteness.

Solution Since you know the total number of iterations, you can use a for-loop. Inside the loop, however, a decision control structure must check whether or not the user-provided number is positive; if so, it must

accumulate the user-provided number in variable s. The variable count counts the number of positive numbers entered. When the flow of execution exits the loop, the average value can then be calculated. The Java program is as follows.

Project_26.1-10

```
public static void main(String[] args) throws Exception {
   int count, i; double s, x;
   s = 0;
   count = 0;
   for (i = 1; i <= 100; i++) {
      x = Double.parseDouble(cin.nextLine());
      if (x > 0) {
         s = s + x;
         count++;
      }
   }
   if (count != 0) {
      System.out.println(s / count);
   }
   else {
      System.out.println("No positive numbers entered!");
   }
}
```

✎ *The* `if (count != 0)` *statement is necessary, because there is a possibility that the user may enter negative values (or zeros) only. By including this check, the program prevents any division-by-zero errors and thereby satisfies the property of definiteness.*

Exercise 26.1-11 Counting the Vowels

Write a Java program that prompts the user to enter a message and then counts and displays the number of vowels the message contains.

Solution The Java program that follows counts the vowels in an English message. The for-loop iterates for all the characters that the message contains. The single-alternative decision structure checks one character at each iteration and if it is a vowel, variable count is increased by one.

Project_26.1-11

```
public static void main(String[] args) throws Exception {
   String message; char character; String vowels = "AEIOU"; int i, count;
```

```
System.out.print("Enter an English message: "); message =
cin.nextLine().toUpperCase();
count = 0;
for (i = 0; i <= message.length() - 1; i++) {
   character = message.charAt(i);
   if (vowels.indexOf(character) != -1) { //If character is found in vowels
      count++;
   }
}
System.out.println("Vowels: " + count); }
```

The `length()` *method returns the number of characters variable* `message` *consists of, whereas the* `message.charAt(i)` *method returns the character located at the specified position (indicated by the variable* `i`*) in the string variable* `message` *(see Section 14.3).*

The `indexOf()` *method returns the value of −1 if* `character` *is not found in variable* `vowels` *(see Section 14.3).*

26.2 Rules that Apply to For-Loops

There are certain rules you must always follow when writing programs with for-loops, since they can save you from undesirable side effects.

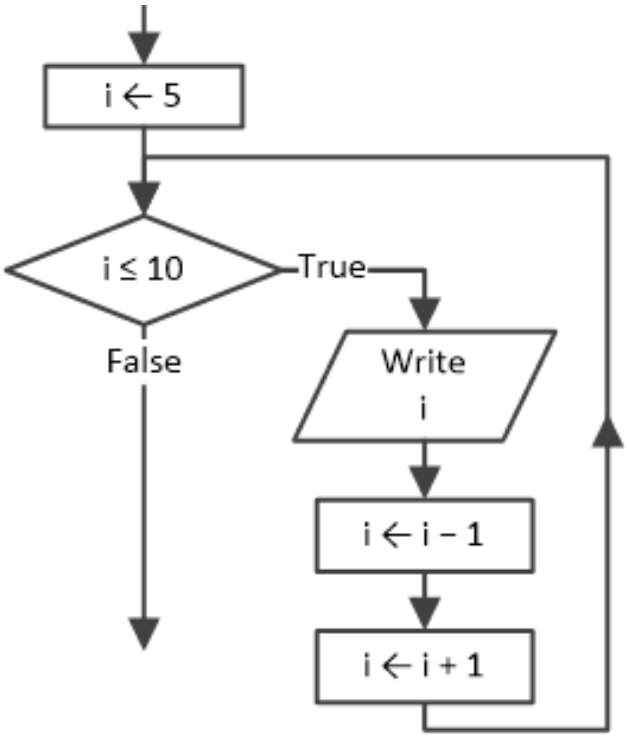
- **Rule 1**: The *counter* variable can appear in a statement inside the loop but its value must never be altered (see Exercise 26.2-1 that follows). The same applies to *final_value* and *offset* in case they are variables and not constant values.
- **Rule 2**: The *offset* must never be zero. If it is set to zero, the loop performs an infinite number of iterations (see Exercise 26.2-2 that follows).
- **Rule 3**: If *initial_value* is smaller than *final_value* then, the *offset* must be positive. If it is negative, the loop performs zero iterations (see Exercise 26.2-3 that follows). Violating this rule on purpose, however, can be useful in certain situations.
- **Rule 4**: If *initial_value* is greater than *final_value* then, the *offset* must be negative. If it is positive, the loop performs zero iterations (see Exercise 26.2-4 that follows). Violating this rule on purpose, however, can be useful in certain situations.

Exercise 26.2-1 Counting the Total Number of Iterations

How many iterations does the following code fragment perform?

```
for (i = 5; i <= 10; i++) {
   System.out.println(i); i--;
}
```

Solution This code fragment violates the first rule of for-loops, which states, the `counter` variable can appear in a statement inside the loop but its value must never be altered. The corresponding flowchart fragment, shown below, can help you better understand what really goes on.



As you can see, since the initial value 5 of variable `i` is less than 10, the flow of execution enters the loop. Inside the loop, however, the statement `i ← i - 1` eliminates the statement `i ← i + 1` and this results in a non-incrementing variable `i`, which can never reach *`final_value`* 10. Thus, the loop performs an infinite number of iterations.

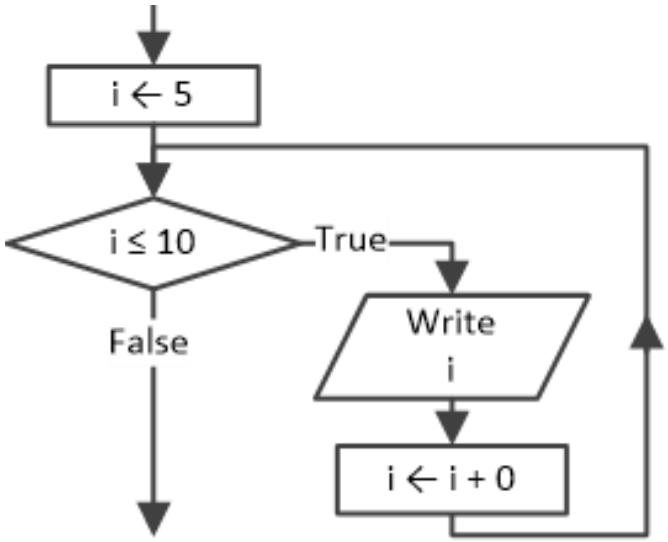
Similar to `counter` variable, if `final_value` and `offset` are variables and not constant values, their value must never alter inside the loop.

Exercise 26.2-2 Counting the Total Number of Iterations

How many iterations does the following code fragment perform?

```
for (i = 5; i <= 10; i += 0) {
   System.out.println(i); }
```

***Solution** This code fragment violates the second rule of for-loops, which states, the `offset` must never be zero. The corresponding flowchart fragment that follows can help you better understand what really goes on.*



As you can see, since the initial value of variable `i` is less than 10, the flow of execution enters the loop. However, the statement `i ← i + 0` never increments variable `i`, which means that it can never reach *`final_value`* 10. Thus, the loop performs an infinite number of iterations.

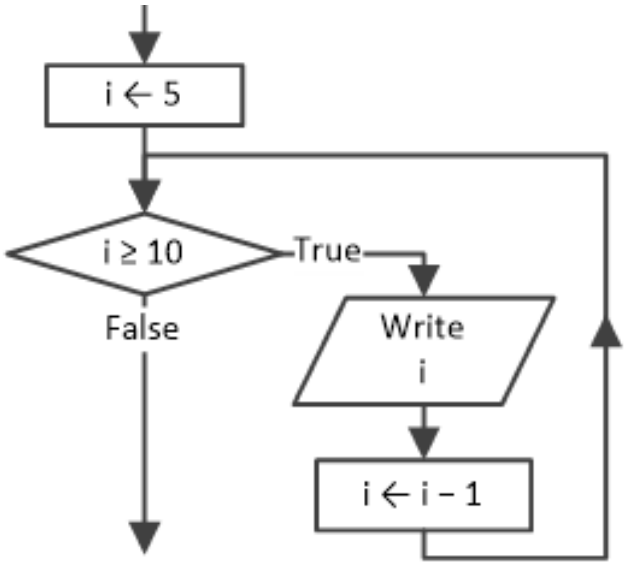
> *The `offset` must never be zero.*

Exercise 26.2-3 Counting the Total Number of Iterations

How many iterations does the following code fragment perform?

```
for (i = 5; i >= 10; i--) {
   System.out.println(i); }
```

***Solution** This code fragment violates the third rule of for-loops, which states, if `initial_value` is smaller than `final_value` then, the `offset` must be positive. If it is negative, the loop performs zero iterations. The corresponding flowchart fragment that follows can help you better understand what really goes on.*



The comparison operator should be ≥ when `counter` decrements and ≤ when `counter` increments.

When the flow of execution reaches the loop control structure, the value 5 is assigned to variable `i`. However, the Boolean expression evaluates to `false` and the flow of execution never enters the loop! Thus, the loop performs zero iterations.

A for-loop is actually a pre-test loop structure. Because of this, it may perform from zero to many iterations.

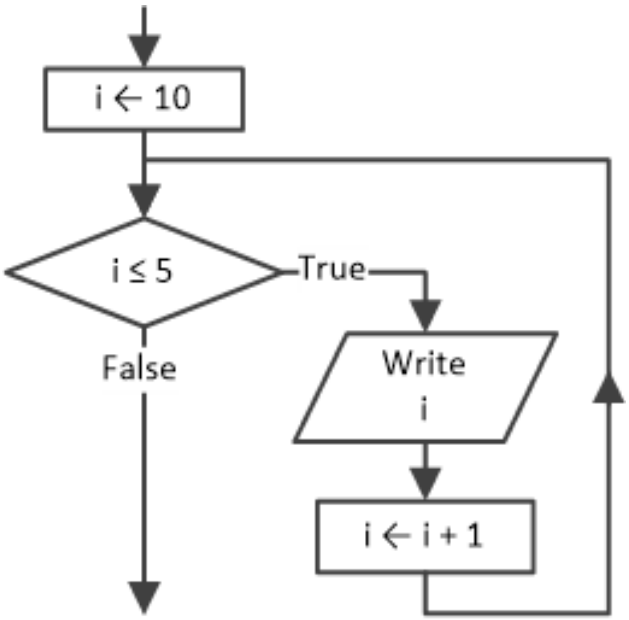
Purposely violating the third rule of for-loops can be useful in certain situations.

Exercise 26.2-4 Counting the Total Number of Iterations

How many iterations does the following code fragment perform?

```
for (i = 10; i <= 5; i++) {
   System.out.println(i); }
```

Solution This code fragment violates the fourth rule of for-loops, which states, if `initial_value` is greater than `final_value` then, the `offset` must me negative. If it is positive, the loop performs zero iterations. The corresponding flowchart fragment that follows can help you better understand what really goes on.



The comparison operator should be ≤ when `counter` increments and ≥ when `counter` decrements.

When the flow of execution reaches the loop control structure, the value 10 is assigned to variable `i`. However, the Boolean expression evaluates to `false` and the flow of execution never enters the loop! Thus, the loop performs zero iterations.

Purposely violating the fourth rule of for-loops can be useful in certain situations.

Exercise 26.2-5 Finding the Sum of N Numbers

Write a Java program that prompts the user to enter N numbers and then calculates and displays their sum. The value of N must be provided by the user at the beginning of the program.

Solution The solution is presented here.

Project_26.2-5

```
public static void main(String[] args) throws Exception {
   int n, i; double a, total;
   System.out.print("Enter quantity of numbers to enter: "); n =
   Integer.parseInt(cin.nextLine());
   total = 0;
   for (i = 1; i <= n; i++) {
      System.out.print("Enter number No " + i + ": ");
```

```
    a = Double.parseDouble(cin.nextLine());
    total += a; //This is equivalent to total = total + a
}
System.out.println("Sum: " + total); }
```

✎ *Even though it violates the fourth rule of for-loops, in this particular exercise this situation is very useful. If the user enters a non-positive value for variable* `n`*, the* `for` *statement performs zero iterations.*

26.3 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) In a `for` statement, the value of *`counter`* increments or decrements automatically at the end of each loop.
2) A definite loop can be used when the number of iterations is known.
3) In a definite loop, the statement or block of statements of the loop is executed at least one time.
4) In a for-loop, the *`initial_value`* cannot be greater than the *`final_value`*.
5) When flow of execution exits a for-loop, the value of *`counter`* is equal to *`final_value`*.
6) In a for-loop, the value of *`initial_value`*, *`final_value`* and *`offset`* can be either an integer or a float.
7) In a for-loop, when *`offset`* is set to zero the loop performs zero iterations.
8) In a for-loop, the *`counter`* variable can appear in a statement inside the loop but its value must never be altered.
9) In a for-loop, the *`offset`* can be zero for certain situations.
10) In the following code fragment, the word “Hello” is displayed 10 times.

```
for (i = 0; i <= 10; i++) {
   System.out.println("Hello"); }
```

11) The following code fragment satisfies the property of finiteness.

```
b = Integer.parseInt(cin.nextLine()); for (i = 0; i <= 8; i += b) {
   System.out.println("Hello"); }
```

12) The following code fragment satisfies the property of definiteness.

```
int i;
for (i = -10; i <= 10; i++) {
   System.out.println(Math.sqrt(i)); }
```

26.4 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) A definite loop that uses the `for` statement
   a) executes one iteration more than the equivalent pre-test loop structure (that uses the `while` statement).
   b) executes one iteration less than the equivalent pre-test loop structure (that uses the `while` statement).
   c) none of the above

2) A definite loop that uses the `for` statement can be used in a problem in which
   a) the user enters numbers repeatedly until the value −1 is entered.
   b) the user enters numbers repeatedly until the value entered is greater than *`final_value`*.
   c) all of the above
   d) none of the above

3) In a for-loop *`initial_value`*, *`final_value`*, and *`offset`* can be
   a) a constant value.
   b) a variable.
   c) an expression.
   d) all of the above

4) In a for-loop, when *`final_value`* and *`offset`* are variables, their values
   a) cannot change inside the loop.
   b) must not change inside the loop.
   c) none of the above

5) In a for-loop, when *`counter`* increments, the *`offset`* is
   a) greater than zero.
   b) equal to zero.
   c) less than zero.
   d) none of the above

6) In a for-loop, the initial value of *`counter`*
   a) must be 0.
   b) can be 0.
   c) cannot be a negative one.
   d) none of the above

7) In a for-loop, variable *`counter`* updates automatically
   a) at the end of each iteration.
   b) at the beginning of each iteration.
   c) It does not update automatically.
   d) none of the above

8) In the following code fragment

```
i = 1;
for (i = 5; i <= 5; i++) {
    System.out.println("Hello Hera"); }
```

the message “Hello Hera” is displayed a) 5 times.

b) 1 time.

c) 0 times.

d) none of the above 9) In the following code fragment

```
for (i = 5; i <= 4; i++) {
    i = 1;
    System.out.println("Hello Artemis"); }
```

the message “Hello Artemis” is displayed a) 1 time.

b) an infinite number of times.

c) 0 times.

d) none of the above 10) In the following code fragment

```
for (i = 5; i <= 5; i++) {
    i = 1;
    System.out.println("Hello Ares"); }
```

the message “Hello Ares” is displayed a) 1 time.

b) an infinite number of times.

c) 0 times.

d) none of the above 11) In the following code fragment

```
for (i = 2; i <= 8; i++) {
    if (i % 2 == 0) {
        System.out.println("Hello Demeter");
    }
}
```

the message “Hello Demeter” is displayed a) 8 times.

b) 7 times.

c) 5 times.

d) none of the above 12) In the following code fragment

```
double i; for (i = 4; i <= 5; i += 0.1) {
    System.out.println("Hello Dionysus"); }
```

the message “Hello Dionysus” is displayed a) 1 time.

b) 2 times.

c) 10 times.

d) 11 times.

13) In the following code fragment

```
k = 0;
for (i = 1; i <= 5; i += 2) {
      k = k + i;
}
System.out.println(k);
```

the value displayed is a) 3.

b) 6.

c) 9.

d) none of the above 14) In the following code fragment

```
k = 0;
for (i = 10; i >= -10; i -= 5) {
      k = k + i;
}
System.out.println(i);
```

the value displayed is a) 0.

b) −10.

c) −15.

d) none of the above

26.5 Review Exercises

Complete the following exercises.

1) Create a trace table to determine the values of the variables in each step of the next code fragment. How many iterations does this code perform?

```
int a, b, j;
a = 0;
b = 0;
for (j = 0; j <= 8; j += 2) {
   if (j < 5) {
      b++;
   }
   else {
```

```
      a += j - 1;
   }
}
System.out.println(a + ", " + b);
```

2) Create a trace table to determine the values of the variables in each step of the next code fragment for two different executions.

The input values for the two executions are: (i) 10, and (ii) 21.

```
int a, b, j;
a = Integer.parseInt(cin.nextLine()); b = a;
for (j = a - 5; j <= a; j += 2) {
   if (j % 2 != 0) {
      b = a + j + 5;
   }
   else {
      b = a - j;
   }
}
System.out.println(b);
```

3) Create a trace table to determine the values of the variables in each step of the next code fragment for the input value 12.

```
int a, j, x, y;
a = Integer.parseInt(cin.nextLine()); for (j = 2; j <= a - 1; j += 3) {
   x = j * 3 + 3; y = j * 2 + 10; if (y - x > 0 || x > 30) {
      y *= 2;
   }
   x += 4;
   System.out.println(x + ", " + y); }
```

4) Fill in the gaps in the following code fragments so that all loops perform exactly five iterations.

i)

```
int a; for (a = 5; a <= …… ; a++) {
      System.out.println(b); b += a;
}
```

ii)

```
double a; for (a = 0; a <= …… ; a += 0.5) {
      System.out.println(b); b += a;
}
```

iii)

```
int a; for (a = …… ; a >= -15; a -= 2) {
```

```
        System.out.println(b); b += a;
    }
```

iv)

```
    int a; for (a = -11 ; a >= -15; a = a ……) {
        System.out.println(b); b += a;
    }
```

5) Without using a trace table, can you find out what the next code fragment displays?

```
String word = "Zeus"; String s = ""; int i;
for (i = word.length() - 1; i >= 0; i--) {
   s = s + word.charAt(i); }
System.out.println(s);
```

6) Design a flowchart and write the corresponding Java program that prompts the user to enter 20 numbers and then calculates and displays their product and their average value.

7) Write a Java program that calculates and displays the sine of all numbers from 0 to 360^{o}, using a step of 0.5. It is given that $2\pi = 360^{o}$.

8) Write a Java program that prompts the user to enter a number in degrees and then calculates and displays the cosine of all numbers from 0 to that user-provided number, using a step of 1. It is given that $2\pi = 360^{o}$.

9) Write a Java program that calculates and displays the sum of the following: $S = 1 + 3 + 5 + \ldots + 99$

10) Write a Java program that lets the user enter an integer N and then calculates and displays the product of the following: $P = 2^{1} \times 4^{3} \times 6^{5} \times \ldots \times 2N^{(2N-1)}$

11) Write a Java program that calculates and displays the sum of the following: $S = 1 + 2 + 4 + 7 + 11 + 16 + 22 + 29 + 37 + \ldots + 191$

12) Design a flowchart and write the corresponding Java program that lets a teacher enter the total number of students as well as their grades and then calculates and displays the average value of those who got an “A”, that is 90 to 100. Add all necessary checks to make the program satisfy the property of definiteness.

13) Design a flowchart and write the corresponding Java program that prompts the user to enter 30 four-digit integers and then calculates and displays the sum of those with a first digit of 5 and a last digit of 3. For example, values 5003, 5923, and 5553 are all such integers.

14) Design a flowchart and write the corresponding Java program that prompts the user to enter N integers and then displays the total number of those that are even. The value of N must be provided by the user at the beginning of the program. Moreover, if all user-provided integers are odd, the message "You entered no even integers" must be displayed.

15) Design a flowchart and write the corresponding Java program that prompts the user to enter 50 integers and then calculates and displays the average value of those that are odd and the average value of those that are even.

16) Design a flowchart and write the corresponding Java program that prompts the user to enter two integers into variables `start` and `finish` and then displays all integers from `start` to `finish`. However, at the beginning the program must check if variable `start` is bigger than variable `finish`. If this happens, the program must swap their values so that they are always in the proper order.

17) Design a flowchart and write the corresponding Java program that prompts the user to enter two integers into variables `start` and `finish` and then displays all integers from `start` to `finish` that are multiples of five. However, at the beginning the program must check if variable `start` is bigger than variable `finish`. If this happens, the program must swap their values so that they are always in the proper order.

18) Write a Java program that prompts the user to enter a real and an integer and then displays the result of the

first number raised to the power of the second number, without using the `Math.pow()` method.

19) Write a Java program that prompts the user to enter a message and then displays the number of words it contains. For example, if the string entered is “My name is Bill Bouras”, the program must display “The message entered contains 5 words”. Assume that the words are separated by a single space character.

    Hint: Use the `length()` method to get the number of characters that the user-provided message contains.

20) Write a Java program that prompts the user to enter a message and then displays the average number of letters in each word. For example, if the message entered is “My name is Aphrodite Boura”, the program must display “The average number of letters in each word is 4.4”. Space characters must not be counted.

21) Write a Java program that prompts the user to enter a message and then counts and displays the number of consonants the message contains.

22) Write a Java program that prompts the user to enter a message and then counts and displays the number of vowels, the number of consonants, and the number of arithmetic characters the message contains.

Chapter 27
Nested Loop Control Structures

27.1 What is a Nested Loop?

A *nested loop* is a loop within another loop or, in other words, an inner loop within an outer one.

The outer loop controls the number of **complete** iterations of the inner loop. This means that the first iteration of the outer loop triggers the inner loop to start iterating until completion. Then, the second iteration of the outer loop triggers the inner loop to start iterating until completion again. This process repeats until the outer loop has performed all of its iterations.

Take the following Java program, for example.

Project_27.1

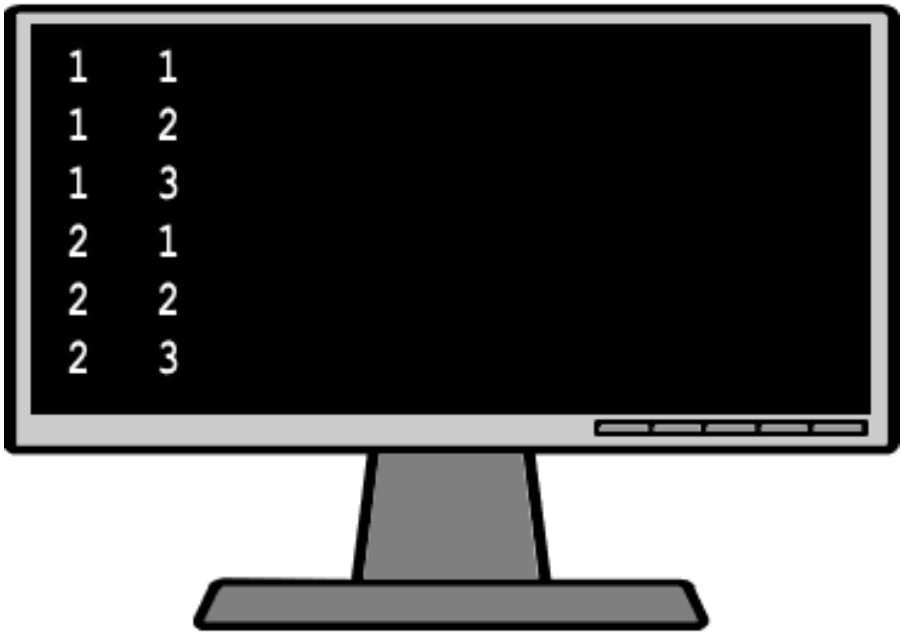
```
public static void main(String[] args) throws Exception {
   int i, j;
   for (i = 1; i <= 2; i++) {
      for (j = 1; j <= 3; j++) { [More…]
         System.out.println(i + " " + j);
      }

   }
}
```

In this program, the outer loop, controlled by the variable `i`, determines the number of complete iterations that the inner loop performs. Specifically, when variable `i` is 1, the inner loop performs three iterations (for `j = 1`, `j = 2`, and `j = 3`). After completing the inner loop, the outer loop needs to perform one more iteration (for `i = 2`). Consequently, the inner loop restarts, performing three new iterations again (for `j = 1`, `j = 2`, and `j = 3`).

The previous example is similar to the following one.

```
i = 1; //Outer loop assigns value 1 to variable i for (j = 1; j <= 3; j++) { //and inner
loop performs three iterations System.out.println(i + " " + j); }
i = 2; //Outer loop assigns value 2 to variable i for (j = 1; j <= 3; j++) { //and inner
loop starts over and performs three new iterations System.out.println(i + " " + j); }
```

The output result is as follows.



✎ *As long as the syntax rules are not violated, you can nest as many loop control structures as you wish. For practical reasons however, as you move to four or five levels of nesting, the entire structure becomes very complex and difficult to understand. However, experience shows that the maximum number of levels of nesting that you will do in your entire life as a programmer is probably three or four.*

✎ *The inner and outer loops do not need to be the same type. For example, a* `for` *statement may nest (enclose) a* `while` *statement, or vice versa.*

Exercise 27.1-1 Say "Hello Zeus". Counting the Total Number of Iterations.

Find the number of times message "Hello Zeus" is displayed.

Project_27.1-1

```
public static void main(String[] args) throws Exception {
   int i, j;
   for (i = 0; i <= 2; i++) {
      for (j = 0; j <= 3; j++) {
         System.out.println("Hello Zeus");
      }
   }
}
```

Solution The values of variables `i` and `j` (in order of appearance) are as follows: ► For `i = 0`, the inner loop performs 4 iterations (for `j = 0`, `j = 1`, `j = 2`, and `j = 3`) and the message "Hello Zeus" is displayed 4 times.

- For `i = 1`, the inner loop performs 4 iterations (for `j = 0`, `j = 1`, `j = 2`, and `j = 3`) and the message “Hello Zeus” is displayed 4 times.
- For `i = 2`, the inner loop performs 4 iterations (for `j = 0`, `j = 1`, `j = 2`, and `j = 3`) and the message “Hello Zeus” is displayed 4 times.

Therefore, the message “Hello Zeus” is displayed a total of 3 × 4 = 12 times.

The outer loop controls the number of complete iterations of the inner one!

Exercise 27.1-2 Creating the Trace Table

For the next code fragment, determine the value that variable `a` contains at the end.

```
int a, i, j; a = 1;
i = 5;
while (i < 7) {
   for (j = 1; j <= 3; j += 2) {
      a = a * j + i;
   }
   i++;
}
System.out.println(a);
```

Solution The trace table is shown here.

Step	Statement	Notes	a	i	j
1	`a = 1`		**1**	?	?
2	`i = 5`		1	**5**	?
3	`i < 7`	This evaluates to `true`			
4	`j = 1`		1	5	**1**
5	`j <= 3`	This evaluates to `true`			
6	`a = a * j + i`		**6**	5	1
7	`j += 2`		6	5	**3**
8	`j <= 3`	This evaluates to `true`			
9	`a = a * j + i`		**23**	5	3

10	`j += 2`		23	5	**5**
11	`j <= 3`	This evaluates to `false`			
12	`i++`		23	**6**	5
13	`i < 7`	This evaluates to `true`			
14	`j = 1`		23	6	**1**
15	`j <= 3`	This evaluates to `true`			
16	`a = a * j + i`		**29**	6	1
17	`j += 2`		29	6	**3**
18	`j <= 3`	This evaluates to `true`			
19	`a = a * j + i`		**93**	6	3
20	`j += 2`		93	6	**5**
21	`j <= 3`	This evaluates to `false`			
22	`i++`		93	**7**	5
23	`i < 7`	This evaluates to `false`			
24	`.println(a)`	It displays: 93			

At the end of the program, variable `a` contains the value 93.

27.2 Rules that Apply to Nested Loops

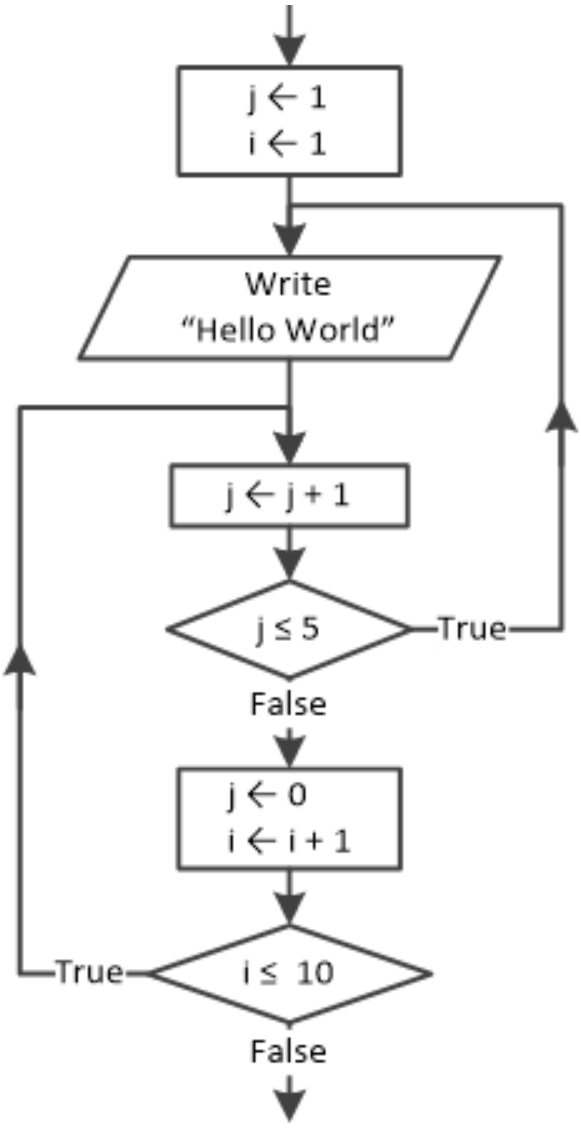
Beyond the four rules that apply to for-loops (presented in Section 26.2), there are two extra rules that you must always follow when writing programs with nested loops, since they can save you from undesirable side effects.

- **Rule 1**: The inner loop must begin and end entirely within the outer loop, which means that the loops must not overlap.
- **Rule 2**: An outer loop and the inner (nested) loop must not use the same *`counter`* variable.

Exercise 27.2-1 Violating the First Rule

Design a flowchart fragment that violates the first rule of nested loops, which states, "The inner loop must begin and end entirely within the outer loop".

Solution The following flowchart fragment violates the first rule of nested loops.



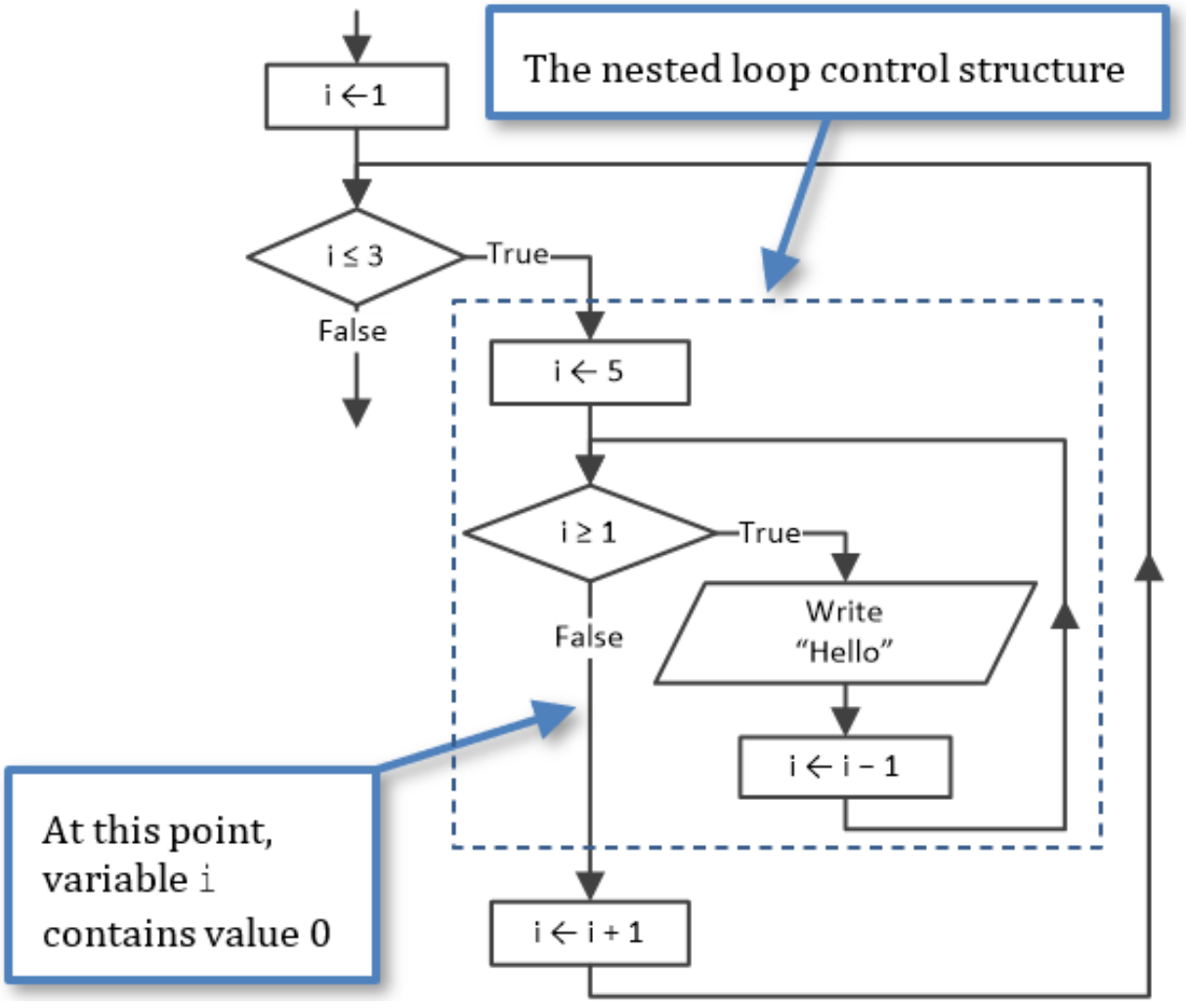
If you try to follow the flow of execution, you will notice that it smoothly performs 5 × 10 = 50 iterations. No one can tell that this flowchart is wrong. In fact, it is technically correct. However, the issue lies in its readability. It's extremely difficult to discern what this flowchart is intended to accomplish. Moreover, this structure matches none of the already familiar loop control structures that you have been taught, so it cannot be directly converted into a Java program as is. Try to avoid this kind of nested loop!

Exercise 27.2-2 Violating the Second Rule

Find the number of times message "Hello" is displayed.

```
for (i = 1; i <= 3; i++) {
   for (i = 5; i >= 1; i--) {
      System.out.println("Hello");
   }
}
```

***Solution** At first glance, one would think that the word "Hello" is displayed 3 × 5 = 15 times. However, a closer second look reveals that things are not always as they seem. This program violates the second rule of nested loops, which states, "An outer loop and the inner (nested) loop must not use the same `counter` variable". Let's design the corresponding flowchart.*



If you try to follow the flow of execution in this flowchart fragment, you can see that when the inner loop completes all of its five iterations, variable `i` contains the value 0. Then, variable `i` increments by 1 and the outer loop repeats again. This process can continue forever since variable `i` can never

exceed the value 3 that the Boolean expression of the outer loop requires. Therefore, the message “Hello” is displayed an infinite number of times.

27.3 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A nested loop is an inner loop within an outer one.
2) It is possible to nest a mid-test loop structure within a pre-test loop structure.
3) The maximum number of levels of nesting in a loop control structure is four.
4) When two loop control structures are nested one within the other, the loop that starts last must complete first.
5) When two loop control structures are nested one within the other, they must not use the same *counter* variable.
6) In the following code fragment, the word “Hello” is displayed six times.

```
for (i = 1; i <= 3; i++) {
   for (j = 1; j <= 3; j++) {
      System.out.println("Hello");
   }
}
```

7) In the following code fragment, the word “Hello” is displayed 12 times.

```
for (i = 0; i <= 1; i++) {
   for (j = 1; j <= 3; j++) {
      for (k = 1; k <= 4; k += 2) {
         System.out.println("Hello");
      }
   }
}
```

8) In the following code fragment, the word “Hello” is displayed an infinite number of times.

```
for (i = 1; i <= 3; i++) {
   for (i = 3; i >= 1; i--) {
      System.out.println("Hello");
   }
}
```

9) In the following code fragment, the word “Hello” is displayed nine times.

```
for (i = 0; i <= 2; i++) {
```

```
   j = 1; do {
      System.out.println("Hello");
      j++;
   } while (j < 4); }
```

10) In the following code fragment, there is at least one mid-test loop structure.

```
int s, a;
s = 0;
while (!false) {
   while (!false) {
      a = Integer.parseInt(cin.nextLine());
      if (a >= -1) break;
   }
   if (a == -1) break; s += a; }
System.out.println(s);
```

27.4 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) In the following code fragment

for (i = 1; i <= 2; i++) {
    for (j = 1; j <= 2; j++) {
        System.out.**println**("*Hello*");
    }
}

the values of variables `i` and `j` (in order of appearance) are a) `j = 1, i = 1, j = 1, i = 2, j = 2, i = 1, j = 2, i = 2`

b) `i = 1, j = 1, i = 1, j = 2, i = 2, j = 1, i = 2, j = 2`

c) `i = 1, j = 1, i = 2, j = 2`

d) `j = 1, i = 1, j = 2, i = 2`

2) In the following code fragment

x = 2;
while (x > -2) {
    do {
        x--;
        System.out.**println**("*Hello Hestia*");
    } **while** (x < -2); }

the message “Hello Hestia” is displayed a) 4 times.

b) an infinite number of times.

c) 0 times.

d) none of the above

3) In the following code fragment

```
x = 1;
while (x != 500) {
    for (i = x; i <= 3; i++) {
        System.out.println("Hello Artemis");
    }
    x++;
}
```

the message “Hello Artemis” is displayed a) an infinite number of times.

b) 1500 times.

c) 6 times.

d) none of the above

4) The following code fragment

```
for (i = 1; i <= 3; i++) {
    for (j = 1; j <= i; j++) {
        System.out.print(i * j + ", ");
    }
}
System.out.print("The End!");
```

displays a) 1, 2, 4, 3, 6, 9, The End!

b) 1, 2, 3, 4, 6, 9, The End!

c) 1, 2, The End!, 4, 3, The End!, 6, 9, The End!

d) none of the above

5) The following code fragment

```
for (i = 1; i <= 10 ; i++) {
    for (i = 10; i >= 1 ; i--) {
        System.out.println("Hello Dionysus");
    }
}
```

does **not** satisfy the property of a) definiteness.

b) finiteness.

c) effectiveness.

d) none of the above

27.5 Review Exercises

Complete the following exercises.

1) Fill in the gaps in the following code fragments so that all code fragments display the message “Hello Hephaestus” exactly 100 times.

i)

```
int a, b; for (a = 6; a < …… ; a++) {
        for (b = 1; b <= 25 ; b++) {
                System.out.println("Hello Hephaestus");
        }
}
```

ii)

```
double a, b; for (a = 0; a <= …… ; a += 0.5) {
        for (b = 10; b <= 19 ; b++) {
                System.out.println("Hello Hephaestus");
        }
}
```

iii)

```
int a; float b; for (a = …… ; a > -17; a -= 2) {
        for (b = 10; b > 0 ; b -= 0.5) {
                System.out.println("Hello Hephaestus");
        }
}
```

iv)

```
int a, b; for (a = -11; a >= -15; a -= 1) {
        for (b = 100; b <= …… ; b += 2) {
                System.out.println("Hello Hephaestus");
        }
}
```

2) Design the corresponding flowchart and create a trace table to determine the values of the variables in each step of the next code fragment.

```
a = 1;
for (j = 1; j <= 2; j += 0.5) {
   i = 10; while (i < 30) {
      a = a + j + i;
      i += 10;
   }
}
System.out.println(a);
```

3) Create a trace table to determine the values of the variables in each step of the next code fragment. How many times is the statement `s = s + i * j` executed?

```
s = 0;
for (i = 1; i <= 4; i++) {
   for (j = 3; j >= i; j--) {
      s = s + i * j;
   }
}
System.out.println(s);
```

4) Create a trace table to determine the values of the variables in each step of the next code fragment for three different executions. How many iterations does this code perform?

   The input values for the three executions are: (i) NO, (ii) YES, NO; and (iii) YES, YES, NO.

```
int s, y, i; String ans;
s = 1;
y = 25;
do {
   for (i = 1; i <= 3; i++) {
      s = s + y;
      y -= 5;
   }
   ans = cin.nextLine(); } while (ans.equals("YES")); System.out.println(s);
```

5) Write a Java program that displays an hours and minutes table in the following form.

   0 0

   0 1

   0 2

   0 3

   ...

   0 59

   1 0

   1 1

   1 2

   ...

   23 59

Please note that the output is aligned with tabs.

6) Using nested loop control structures, write a Java program that displays the following output.

```
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

7) Using nested loop control structures, write a Java program that displays the following output.

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
0 1 2 3 4 5
```

8) Using nested loop control structures, write a Java program that displays the following rectangle.

```
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
* * * * * * * * * * *
```

9) Write a Java program that prompts the user to enter an integer N between 3 and 20 and then displays a square of size N on each side. For example, if the user enters 4 for N, the program must display the following square.

```
* * * *
* * * *
* * * *
* * * *
```

10) Write a Java program that prompts the user to enter an integer N between 3 and 20 and then displays a hollow square of size N on each

side. For example, if the user enters 4 for N, the program must display the following hollow square.

```
* * * *
*   *
*   *
* * * *
```

11) Using nested loop control structures, write a Java program that displays the following triangle.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

Chapter 28
More about Flowcharts with Loop Control Structures

28.1 Introduction

By working through the previous chapters, you have become familiar with all the loop control structures. Since flowcharts are an ideal way to learn "Algorithmic Thinking" and to help you better understand specific control structures, this chapter will teach you how to convert a Java program to a flowchart as well as how to convert a flowchart to a Java program.

28.2 Converting Java Programs to Flowcharts

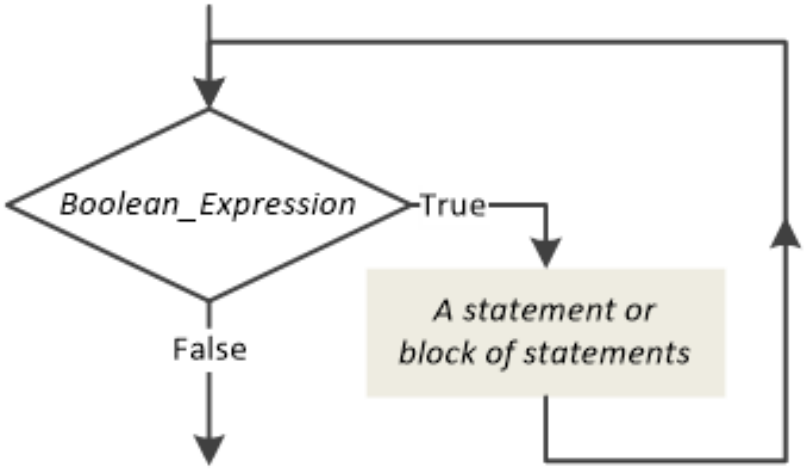
To convert a Java program to a flowchart, you need to recall all loop control structures and their corresponding flowcharts. Following you will find them all summarized.

The Pre-Test Loop Structure

while (*Boolean_Expression*) {

```
    A statement or block of statements
```

}

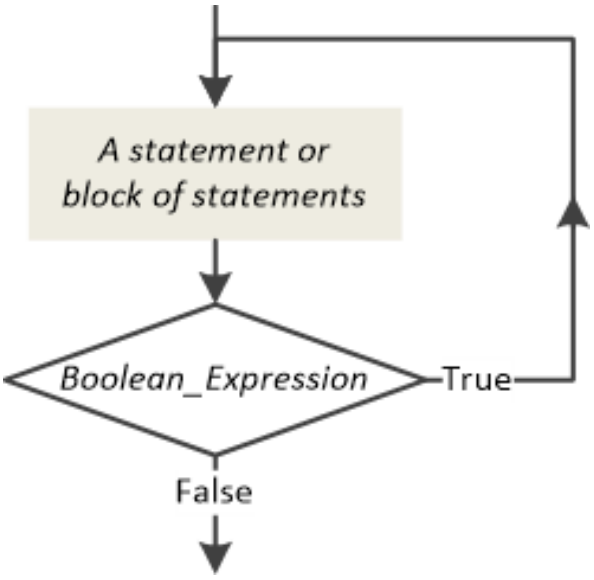


The Post-Test Loop Structure

do {

```
    A statement or block of statements
```

while (*Boolean_Expression*);



The Mid-Test Loop Structure

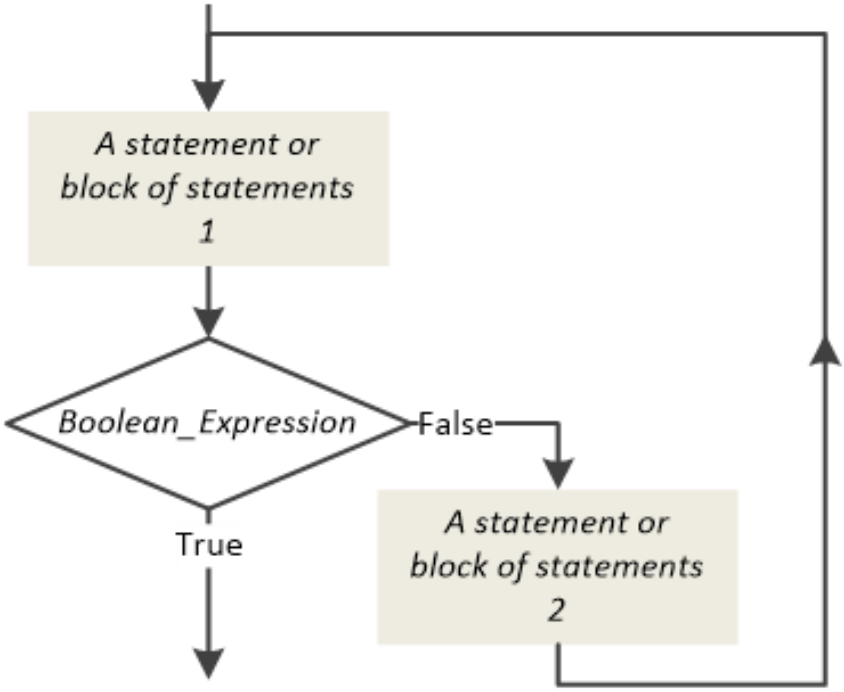
while (true) {

```
A statement or block of statements 1
```

if (*Boolean_Expression*) **break**;

```
A statement or block of statements 2
```

}

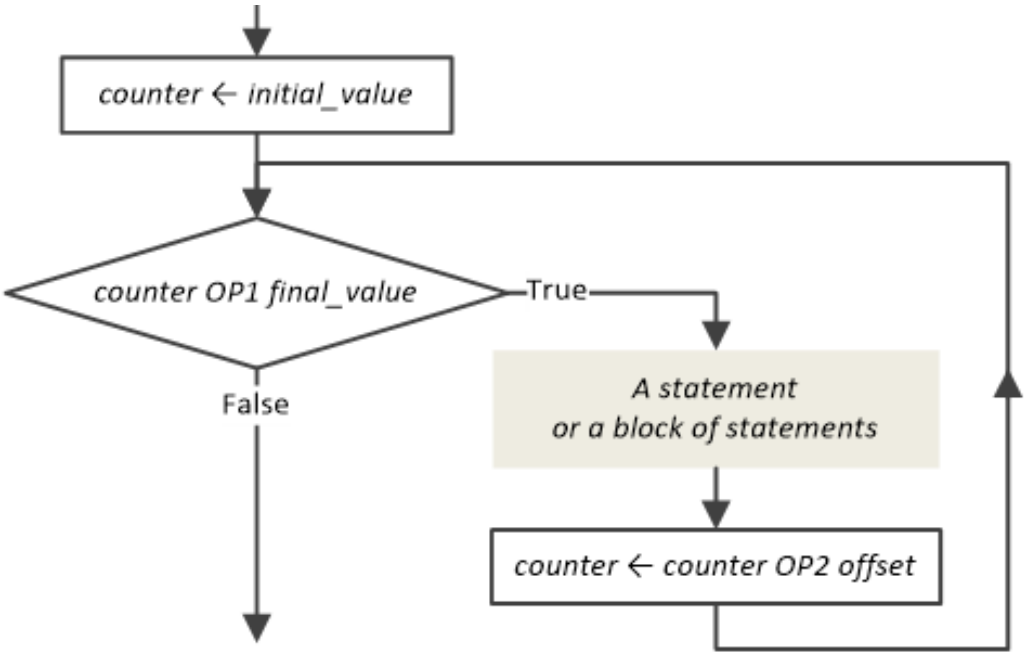


The For-Loop

for (*counter = initial_value*; *counter OP1 final_value*; *counter = counter OP2 offset*) {

```
    A statement or block of statements
```

}



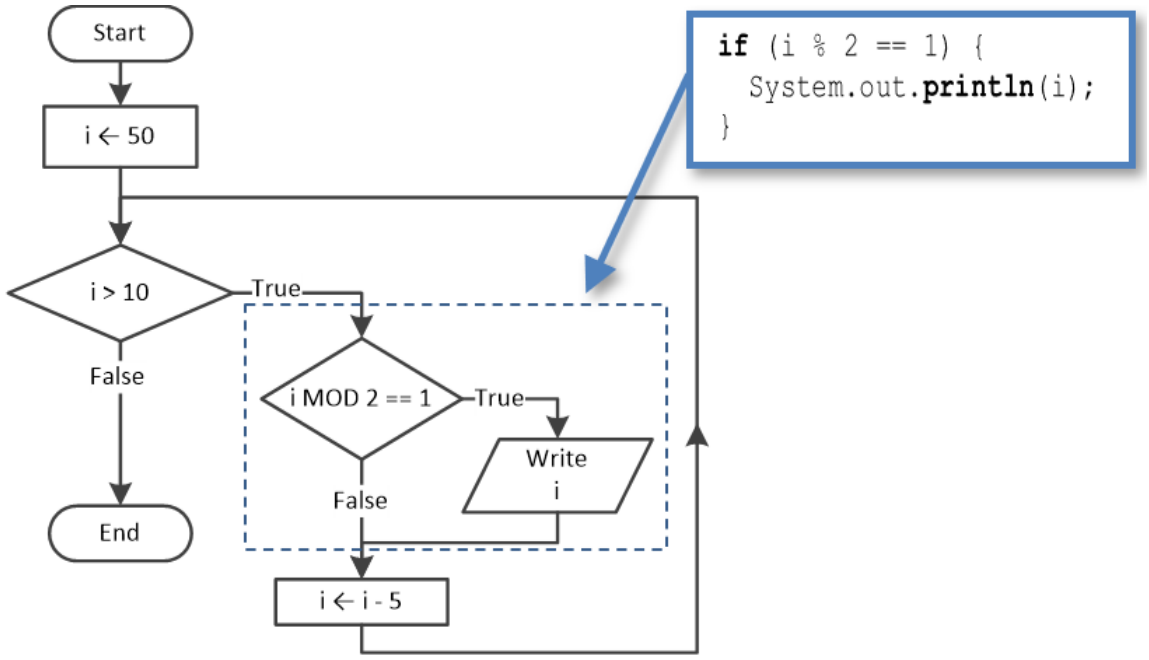
Next, you will find many exercises that can clarify things that you might still need help understanding.

Exercise 28.2-1 Designing the Flowchart Fragment

Design the flowchart that corresponds to the following code fragment.

```
int i = 50; while (i > 10) {
   if (i % 2 == 1) {
      System.out.println(i);
   }
   i -= 5; }
```

Solution This code fragment contains a pre-test loop structure which nests a single-alternative decision structure. The corresponding flowchart fragment that follows includes what you have been taught so far.

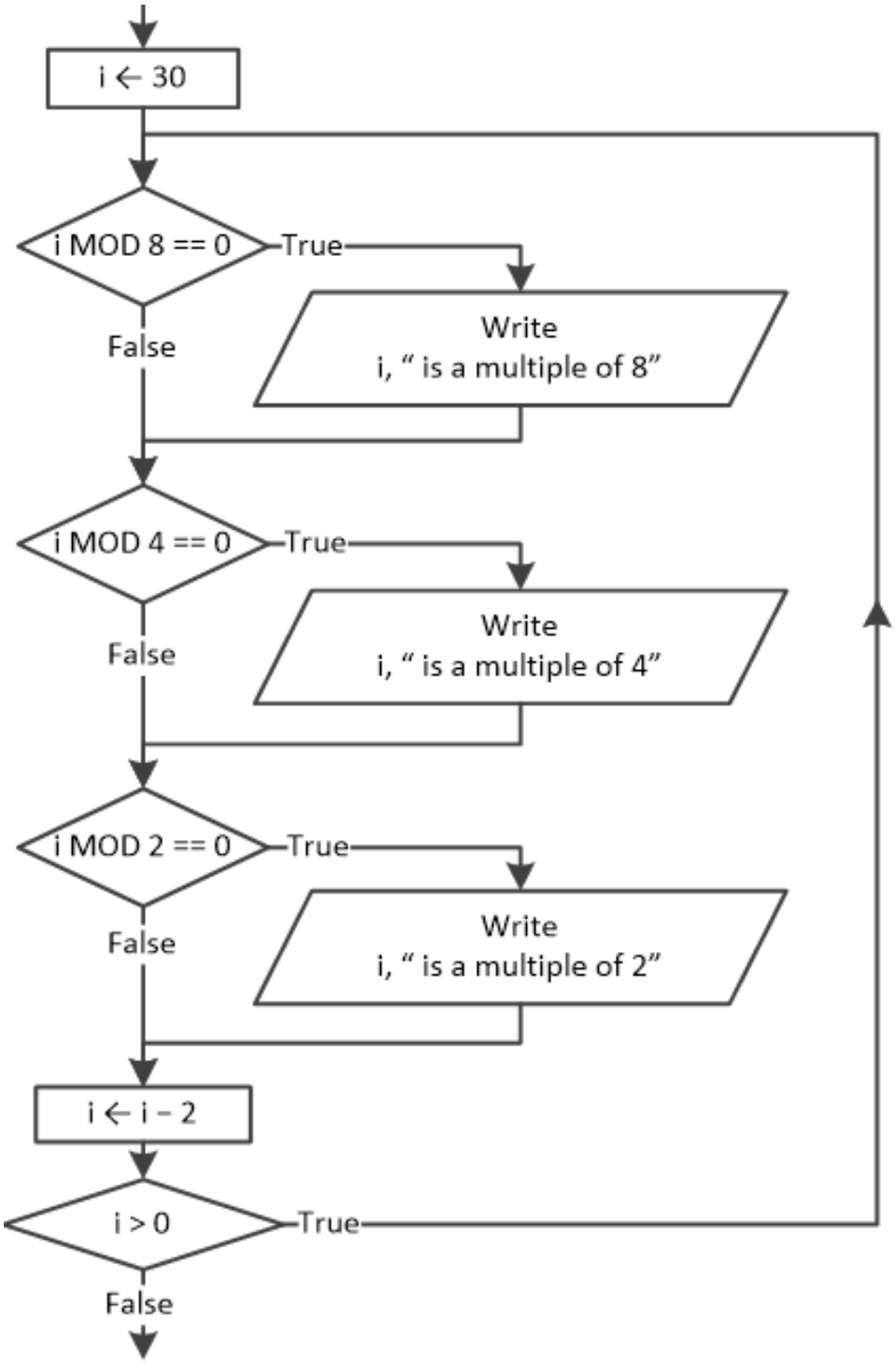


Exercise 28.2-2 Designing the Flowchart Fragment

Design the flowchart that corresponds to the following code fragment.

```
int i = 30; do {
   if (i % 8 == 0) {
      System.out.println(i + " is a multiple of 8");
   }
   if (i % 4 == 0) {
      System.out.println(i + " is a multiple of 4");
   }
   if (i % 2 == 0) {
      System.out.println(i + " is a multiple of 2");
   }
   i -= 2; } while (i > 0);
```

Solution This code fragment contains a post-test loop structure that nests three single-alternative decision structures. The corresponding flowchart fragment is as follows.



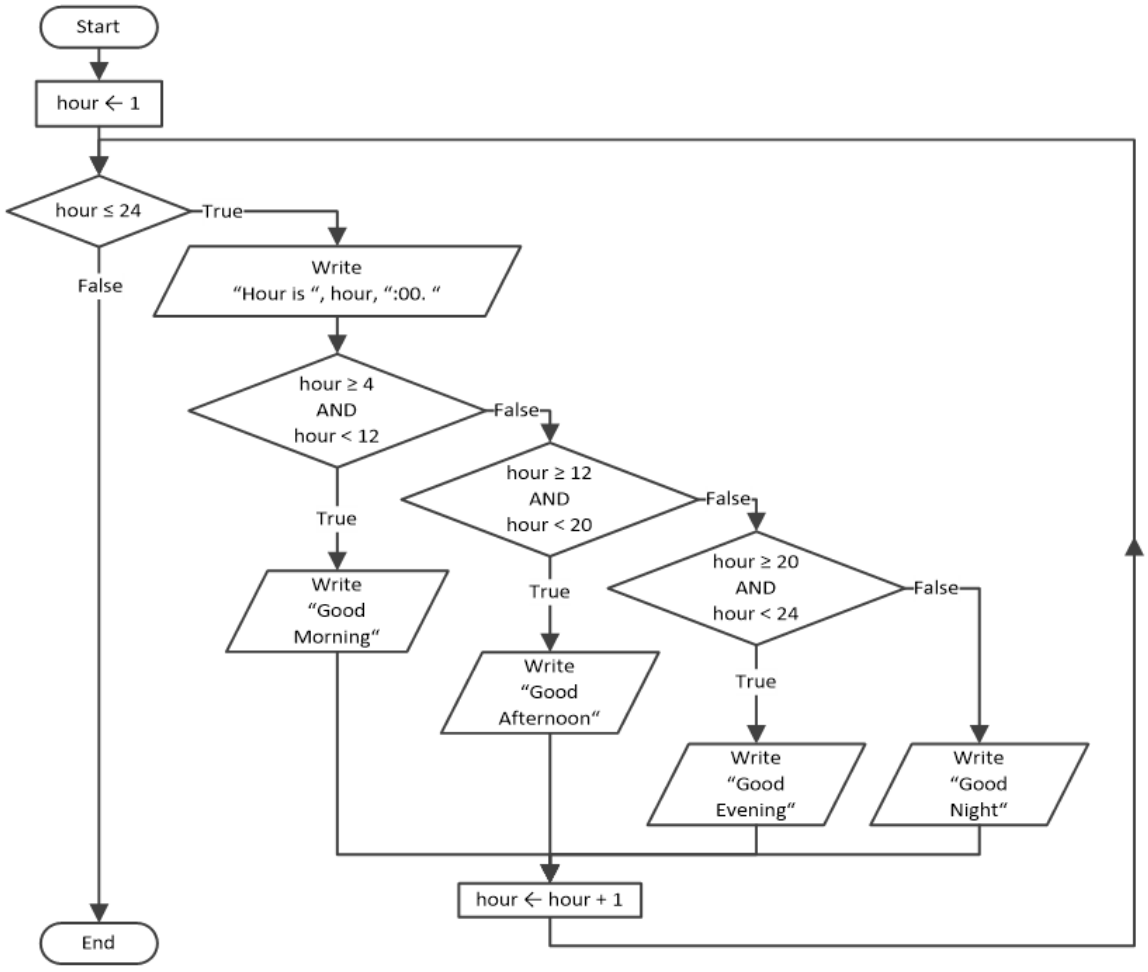
Exercise 28.2-3 Designing the Flowchart

Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int hour;
   for (hour = 1; hour <= 24; hour++) {
      System.out.println("Hour is " + hour + ":00. ");
```

```
      if (hour >= 4 && hour < 12) {
         System.out.println("Good Morning");
      }
      else if (hour >= 12 && hour < 20) {
         System.out.println("Good Afternoon");
      }
      else if (hour >= 20 && hour < 24) {
         System.out.println("Good Evening");
      }
      else {
         System.out.println("Good Night");
      }
   }
}
```

Solution This Java program contains a for-loop that nests a multiple-alternative decision structure. The corresponding flowchart is as follows.

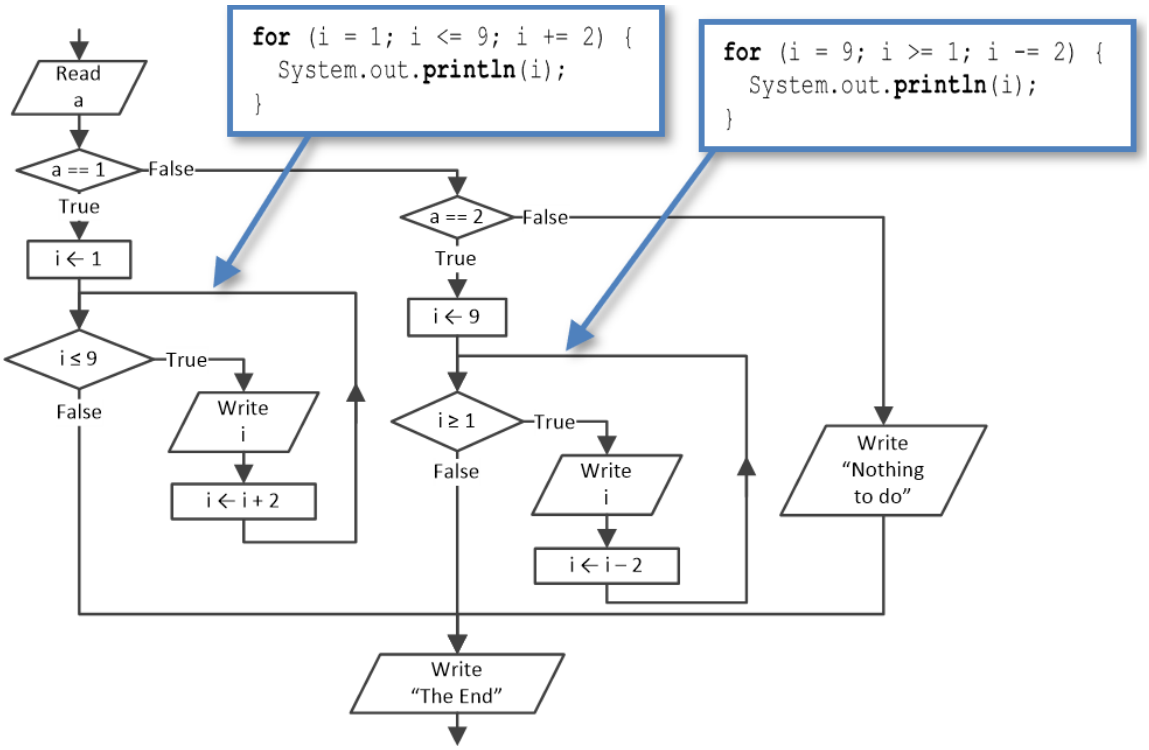


Exercise 28.2-4 Designing the Flowchart Fragment

Design the flowchart that corresponds to the following code fragment.

```
int a, i; a = Integer.parseInt(cin.nextLine());
switch (a) {
   case 1:
   for (i = 1; i <= 9; i += 2) {
         System.out.println(i);
      }
      break;
   case 2:
   for (i = 9; i >= 1; i -= 2) {
         System.out.println(i);
      }
      break;
   default:
   System.out.println("Nothing to do!");
}
System.out.println("The End!");
```

Solution This code fragment contains a case decision structure that nests two for-loops. The corresponding flowchart fragment is as follows.



The multiple-alternative decision structure and the case decision structure can share the same flowchart.

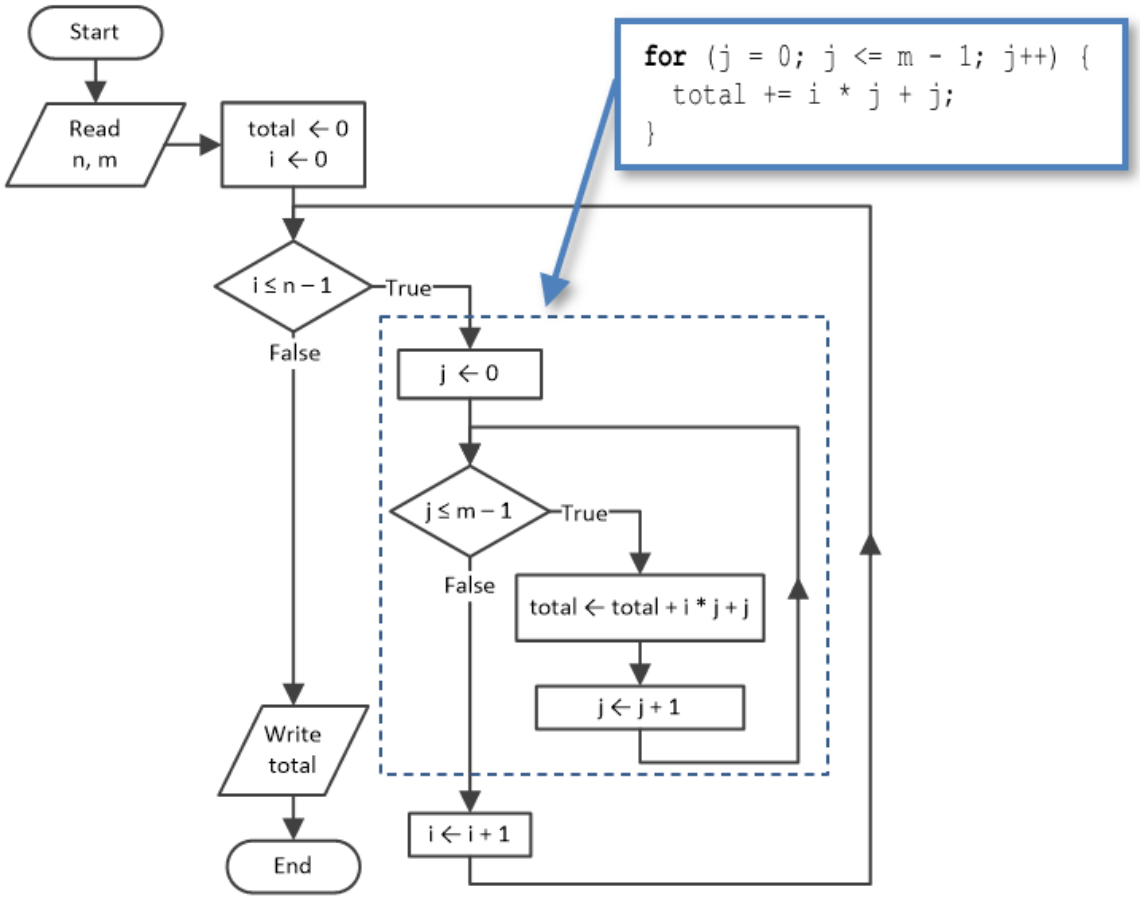
Exercise 28.2-5 Designing the Flowchart

Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
```

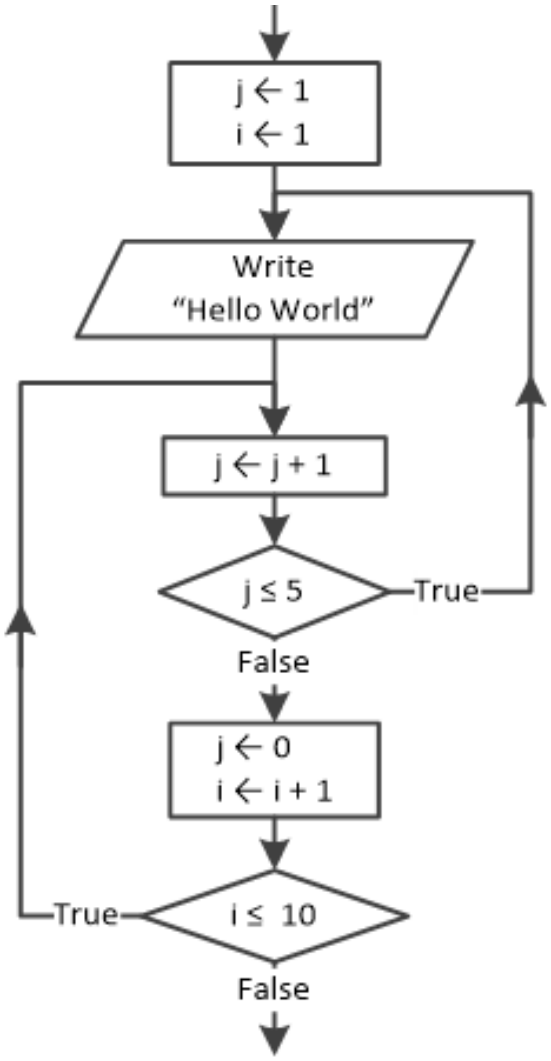
```
int n, m, total, i, j;
n = Integer.parseInt(cin.nextLine()); m = Integer.parseInt(cin.nextLine());
total = 0; for (i = 0; i <= n - 1; i++) {
   for (j = 0; j <= m - 1; j++) {
      total += i * j + j;
   }
}
System.out.println(total); }
```

Solution This Java program contains nested loop control structures; a for-loop nested within another for-loop. The corresponding flowchart is as follows.



28.3 Converting Flowcharts to Java Programs

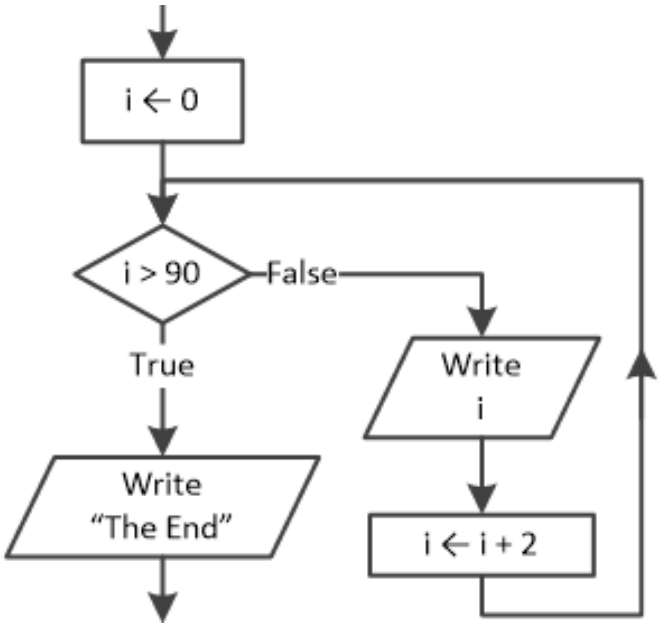
This conversion is not always an easy one. There are cases in which the flowchart designers follow no particular rules, so the initial flowchart may need some modifications before it can be converted into a Java program. The following is an example of one such case.



As you can see, the loop control structures included in this flowchart fragment match none of the structures that you have already learned, such the pre-test, the post-test, the mid-test, or even the for-loop control structure. Thus, you have only one choice and this is to modify the flowchart by adding extra statements or removing existing ones until known loop control structures start to appear. Below are some exercises, and in some of them, the initial flowchart does need modification.

Exercise 28.3-1 Writing the Java Program

Write the Java code that corresponds to the following flowchart fragment.



Solution This is an easy one. The only obstacle you have to overcome is that the true and false paths are not quite in the right position. You need the true and not the false path to actually iterate. As you already know, it is possible to switch the two paths but you need to negate the Boolean expression as well. Thus, the corresponding code fragment becomes

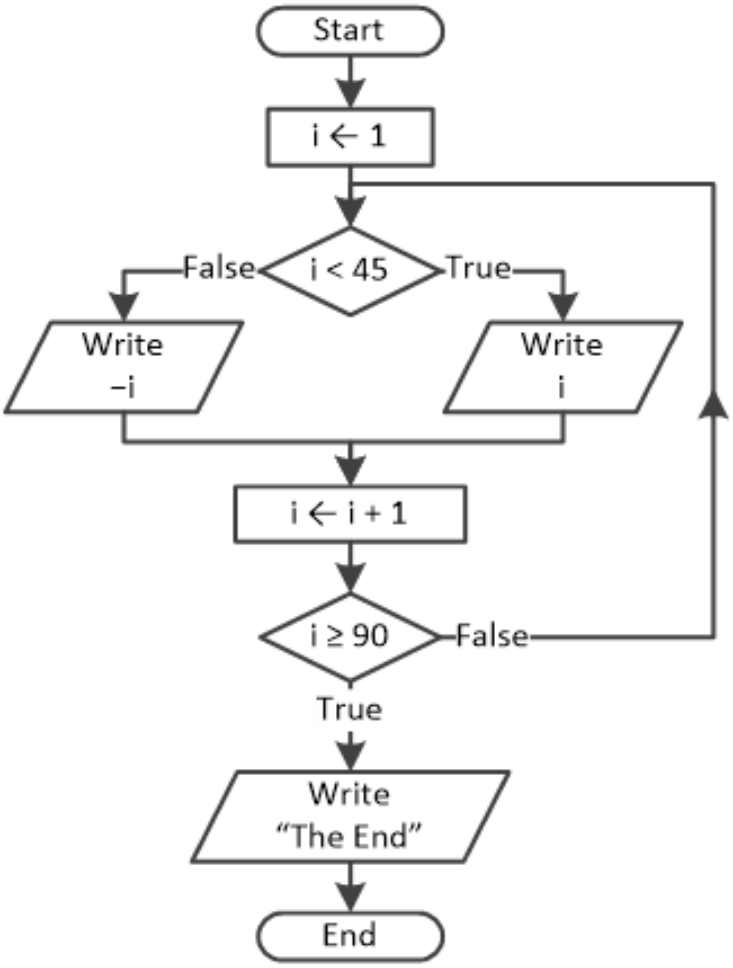
```
i = 0;
while (i <= 90) {
      System.out.println(i); i = i + 2; }
System.out.println("The End");
```

Using a for-loop, this code fragment can equivalently be written as

```
for (i = 0; i <= 90; i += 2) {
      System.out.println(i); }
System.out.println("The End");
```

Exercise 28.3-2 Writing the Java Program

Write the Java program that corresponds to the following flowchart.

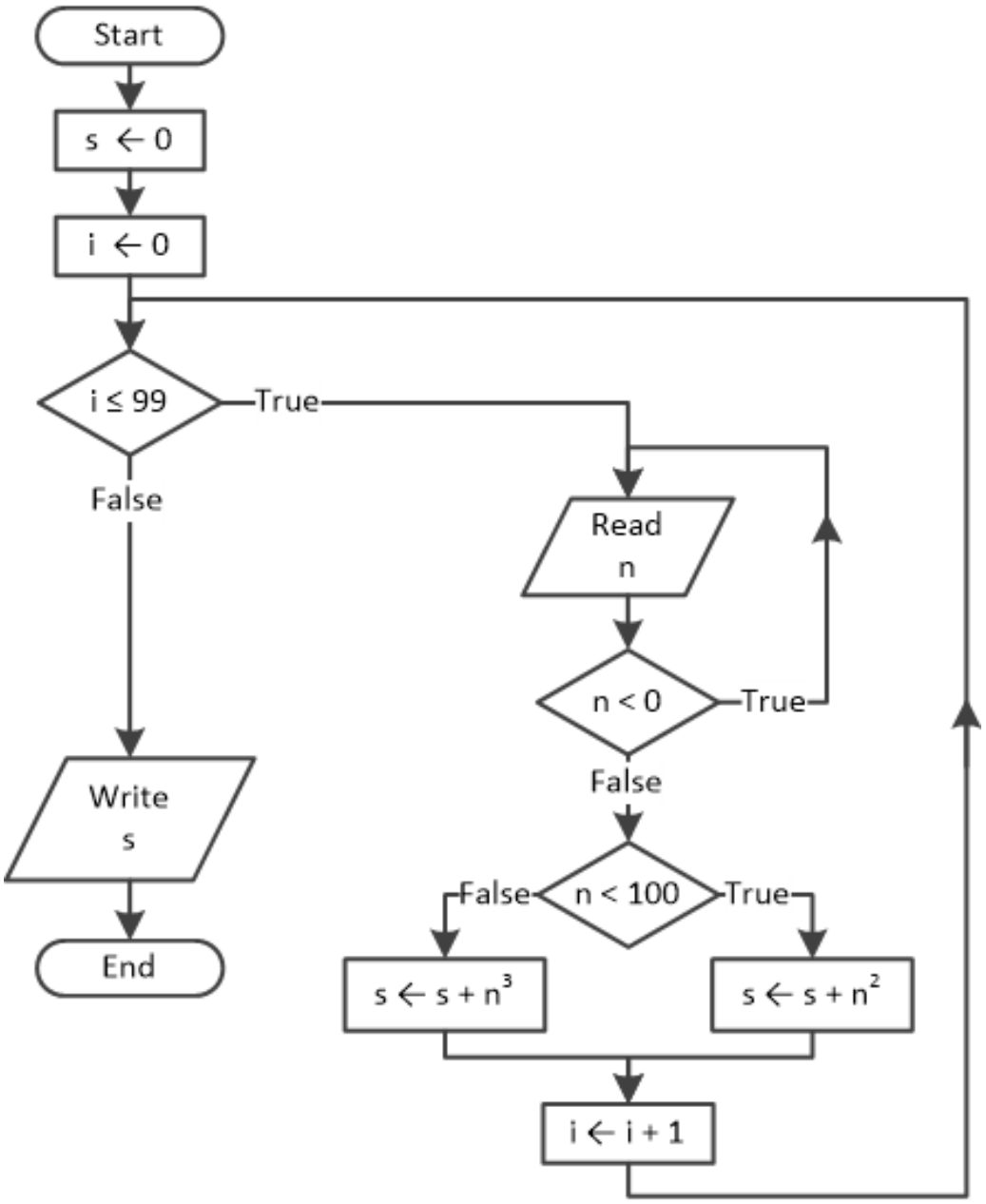


Solution This flowchart contains a post-test loop structure that nests a dual-alternative decision structure. The Java program is as follows.

```
public static void main(String[] args) throws Exception {
   int i;
   i = 1; do {
      if (i < 45) { [More…]
         System.out.println(i);
      }
      else {
         System.out.println(-i);
      }
      i++;
   } while (i < 90);
   System.out.println("The End"); }
```

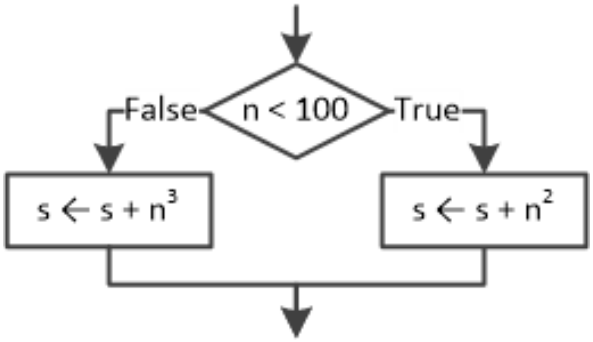
Exercise 28.3-3 Writing the Java Program

Write the Java program that corresponds to the following flowchart.



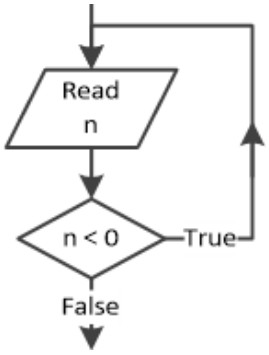
Solution Oops! What a mess! There are so many diamonds here! Be careful, though, as not all of them are decision control structures. In fact, two of them are loop control structures, and only one represents a decision control structure! Can you spot the latter?

You should be quite familiar with loop control structures so far. As you already know, in loop control structures, one of the diamond's (rhombus's) exits always has an upward direction. Thus, the following flowchart fragment, extracted from the initial one, is obviously the decision control structure that you are looking for.

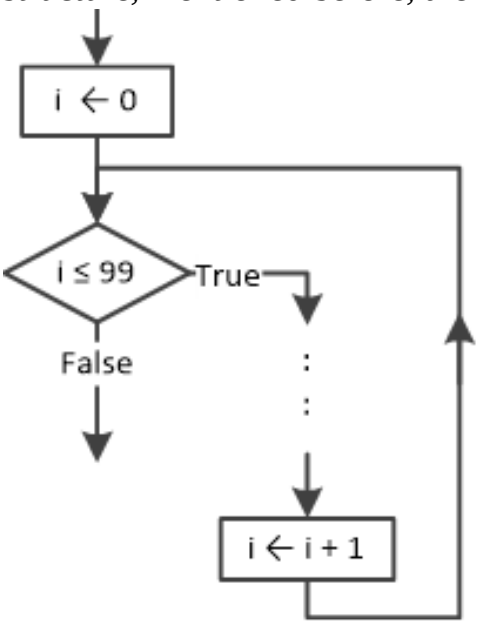


And of course, it's a dual-alternative decision structure!

Now, let's identify the rest of the structures. Right before the dual-alternative decision structure, there is a post-test loop structure. Its flowchart fragment is as follows.



And finally, both the dual-alternative decision structure and the post-test loop structure, mentioned before, are nested within the next flowchart fragment,



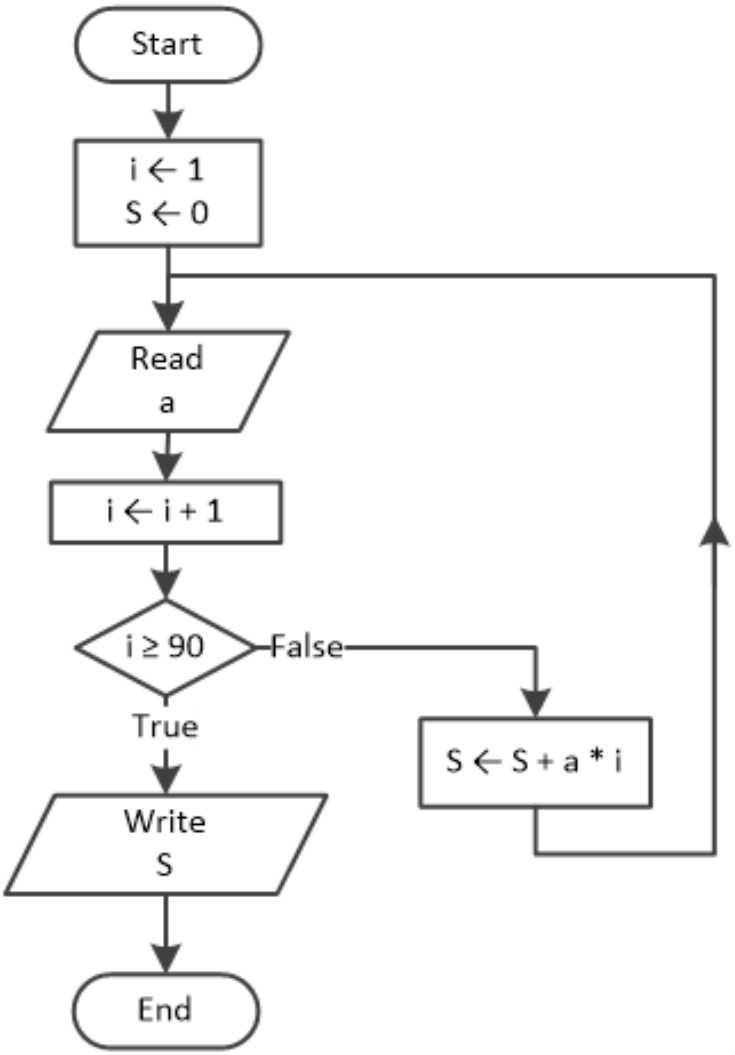
which happens to be a pre-test loop structure and can be written in Java using either a `while` or a `for` statement. The corresponding Java program is as follows.

```
public static void main(String[] args) throws Exception {
   int i; double s, n;
   s = 0; for (i = 0; i <= 99; i++) {
      do { [More…]
         n = Double.parseDouble(cin.nextLine());
      } while (n < 0);
      if (n < 100) [More…]
         s = s + Math.pow(n, 2);
      else
         s = s + Math.pow(n, 3);
   }
   System.out.println(s); }
```

Wasn't so difficult after all, was it?

Exercise 28.3-4 Writing the Java Program

Write the Java program that corresponds to the following flowchart.



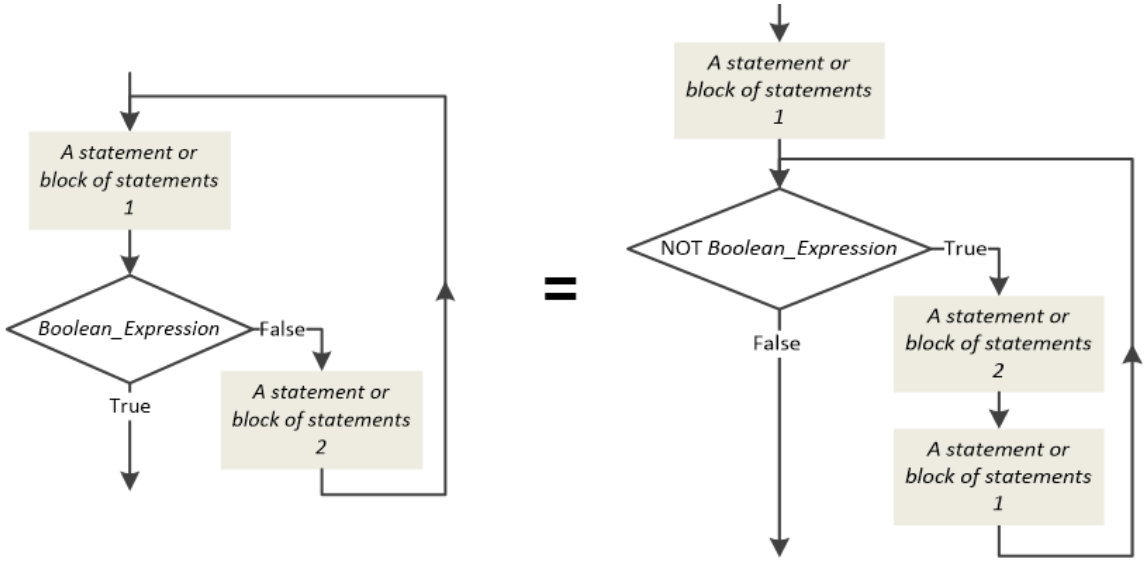
Solution This is a mid-test loop structure. Since there is no direct Java statement for this structure, you can use the `break` statement—or you can even convert the flowchart to something more familiar as shown in the next two approaches First approach – Using the `break` statement The main idea is to create an endless loop `while (true) { ... }` and break out of it when the Boolean expression that exists between the two statements or blocks of statements evaluates to `true` (see Section 25.3).

According to this approach, the initial flowchart can be written in Java as follows.

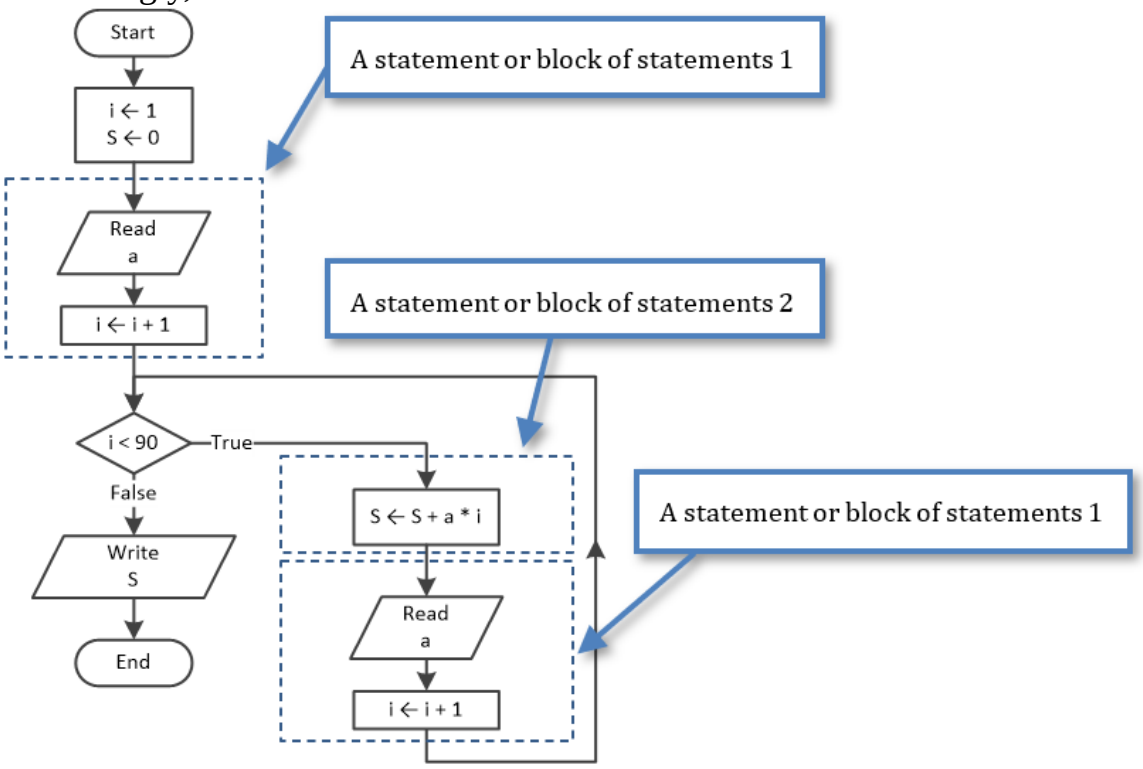
```
public static void main(String[] args) throws Exception {
   int i; double S, a;
   i = 1; S = 0;
   while (true) {
      a = Double.parseDouble(cin.nextLine()); [More…]
      i++;

      if (i >= 90) break;
      S = S + a * i; [More…]

   }
   System.out.println(S); }
```

✎ *Keep in mind that even though the `break` statement can sometimes be useful, it may also lead you to write code that is difficult to read and understand, especially when you make extensive use of it. So, please use it cautiously and sparingly!*

Second approach – Converting the flowchart The mid-test loop structure and its equivalent, using a pre-test loop structure, are as follows.



Accordingly, the initial flowchart becomes



Now, it's easy to write the corresponding Java program.

```
public static void main(String[] args) throws Exception {
   int i; double S, a;
   i = 1; S = 0;
   a = Double.parseDouble(cin.nextLine()); [More…]
   i++;
   while (i < 90) {
      S = S + a * i; [More…]
      a = Double.parseDouble(cin.nextLine()); [More…]
      i++;
   }
   System.out.println(S); }
```

28.4 Review Exercises

Complete the following exercises.

1) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int i, x;
   i = 35; while (i > -35) {
      if (i % 2 == 0)
         System.out.println(2 * i);
```

```
      else
         System.out.println(3 * i);
      i--;
   }
}
```

2) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int i, x;
   i = -20; do {
      x = Integer.parseInt(cin.nextLine());
      if (x == 0)
         System.out.println("Zero");
      else if (x % 2 == 0)
         System.out.println(2 * i);
      else
         System.out.println(3 * i);
      i++;
   } while (i <= 20); }
```

3) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a, i;
   a = Integer.parseInt(cin.nextLine()); if (a > 0) {
      i = 0;
      while (i <= a) {
         System.out.println(i);
         i += 5;
      }
   }
   else {
      System.out.println("Non-Positive Entered!");
   }
}
```

4) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a, i;
   a = Integer.parseInt(cin.nextLine()); if (a > 0) {
      i = 0;
      while (i <= a) {
         System.out.println(3 * i + i / 2.0);
         i++;
      }
   }
   else {
      i = 10;
      do {
         System.out.println(2 * i - i / 3.0);
```

```
         i -= 3;
      } while (i >= a);
   }
}
```

5) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a, b, i;
   a = Integer.parseInt(cin.nextLine()); if (a > 0) {
      for (i = 0; i <= a; i++) {
         System.out.println(3 * i + i / 2.0);
      }
   }
   else if (a == 0) {
      b = Integer.parseInt(cin.nextLine());
      while (b > 0) {
         b = Integer.parseInt(cin.nextLine());
      }
      System.out.println(2 * a + b);
   }
   else {
      b = Integer.parseInt(cin.nextLine());
      while (b < 0) {
         b = Integer.parseInt(cin.nextLine());
      }
      for (i = a; i <= b; i++) {
         System.out.println(i);
      }
   }
}
```

6) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a, b, c, d, total, i, j;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine()); c =
   Integer.parseInt(cin.nextLine()); d = Integer.parseInt(cin.nextLine());
   total = 0; for (i = a; i <= b - 1; i++) {
      for (j = c; j <= d - 1; j += 2) {
         total += i + j;
      }
   }
   System.out.println(total); }
```

7) Design the flowchart that corresponds to the following code fragment.

```
int i; double n, s = 0; for (i = 0; i <= 99; i++) {
   n = Double.parseDouble(cin.nextLine()); while (n < 0) {
      System.out.println("Error");
      n = Double.parseDouble(cin.nextLine());
   }
```

```
   s += Math.sqrt(n); }
System.out.println(s);
```

8) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int i; double s, n;
   s = 0; for (i = 1; i <= 50; i++) {
      do {
         n = Integer.parseInt(cin.nextLine());
      } while (n < 0);
      s += Math.sqrt(n);
   }
   System.out.println(s); }
```

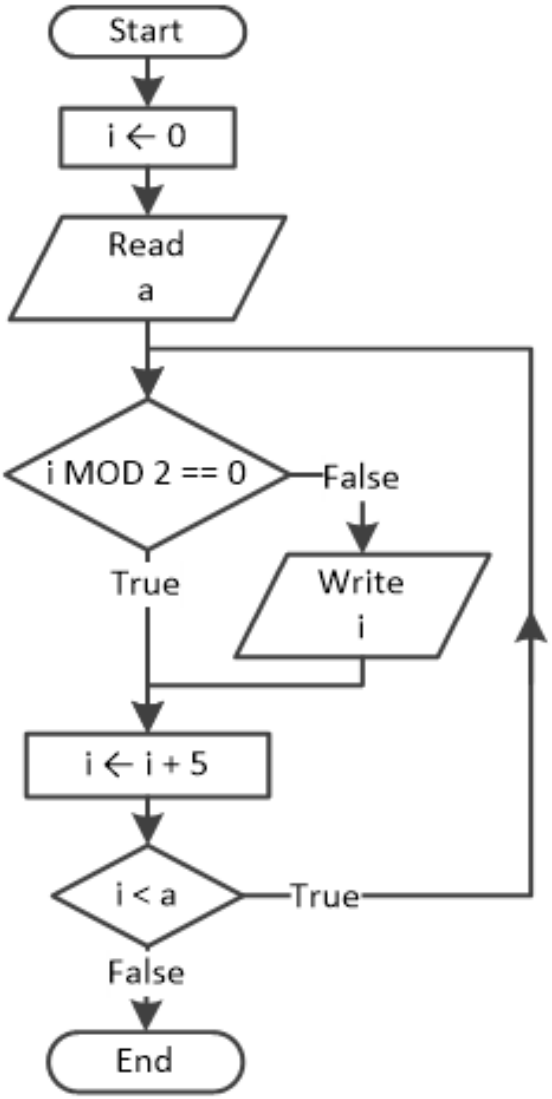
9) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a, b;
   do {
      do {
         a = Integer.parseInt(cin.nextLine());
      } while (a < 0);
      do {
         b = Integer.parseInt(cin.nextLine());
      } while (b < 0);
      System.out.println(Math.abs(a - b));
   } while (Math.abs(a - b) > 100); }
```

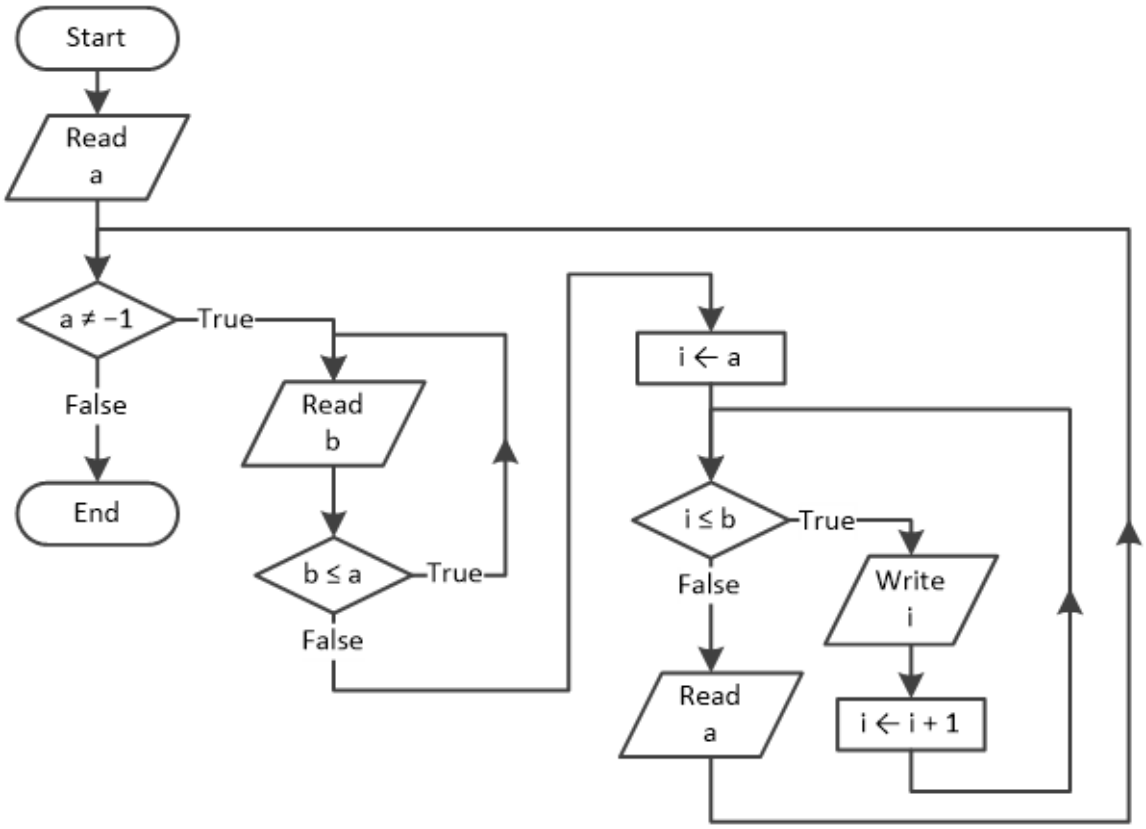
10) Design the flowchart that corresponds to the following Java program.

```
public static void main(String[] args) throws Exception {
   int a, b;
   do {
      do {
         a = Integer.parseInt(cin.nextLine());
         b = Integer.parseInt(cin.nextLine());
      } while (a < 0 || b < 0);
      if (a > b) {
         System.out.println(a - b);
      }
      else {
         System.out.println(a * b);
      }
   } while (Math.abs(a - b) > 100); }
```

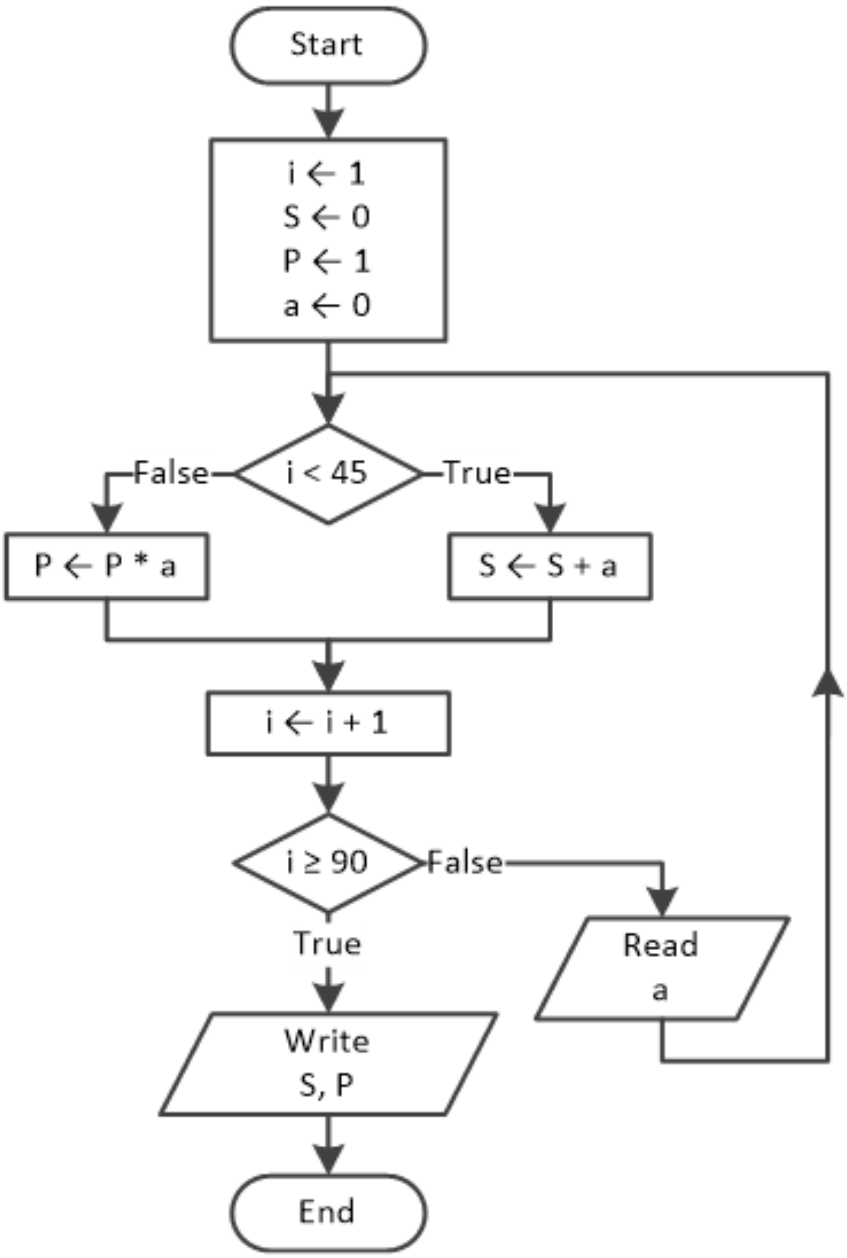
11) Write the Java program that corresponds to the following flowchart.



12) Write the Java program that corresponds to the following flowchart.



13) Write the Java program that corresponds to the following flowchart.


Start
i ← 1
S ← 0
P ← 1
a ← 0
i < 45
False
True
P ← P * a
S ← S + a
i ← i + 1
i ≥ 90
False
True
Read
a
Write
S, P
End

Chapter 29
Tips and Tricks with Loop Control Structures

29.1 Introduction

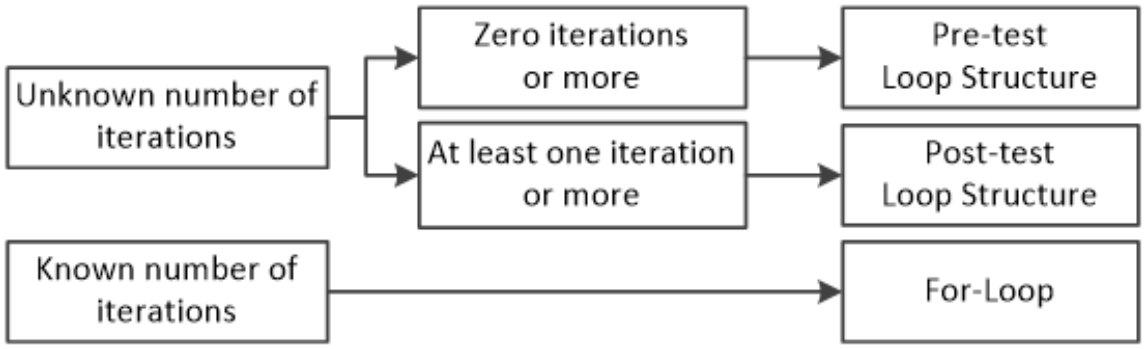
This chapter is dedicated to teaching you some useful tips and tricks that can help you write “better” code. You should always keep them in mind when you design your own algorithms, or even your own Java programs.

These tips and tricks can help you increase your code's readability, help you choose which loop control structure is better to use in each given problem, and help make the code shorter or even faster. Of course there is no single perfect method because on one occasion the use of a specific tip or trick may help, but on another occasion the same tip or trick may have exactly the opposite result. Most of the time, code optimization is a matter of programming experience.

Smaller algorithms are not always the best solution to a given problem. In order to solve a specific problem, you might write a concise algorithm that unfortunately proves to consume a significant amount of CPU time and/or a large portion of main memory (RAM). On the other hand, you might solve the same problem with another algorithm that appears longer but calculates the result much faster and/or utilizes less RAM.

29.2 Choosing a Loop Control Structure

The following diagram can help you choose the most appropriate loop control structure to use in each given problem, depending on the number of iterations.



This diagram recommends the best option, not the only option. For example, when the number of iterations is known, it is not wrong to use a pre-test or a post-test loop structure instead. The proposed for-loop, though, is more convenient.

29.3 The “Ultimate” Rule

One question that often preys on programmers' minds when using pre-test or post-test loop structures, is how to determine which statements should be written inside, and which outside, the loop control structure and in which order.

There is one simple yet powerful rule—the *"Ultimate" rule*! Once you follow it, the potential for making a logic error is reduced to zero!

The "Ultimate" rule states: ► The variable or variables that participate in a loop's Boolean expression must be initialized before entering the loop.

- ► The value of the variable or variables that participate in a loop's Boolean expression must be updated (altered) within the loop. And more specifically, the statement that does this update/alteration must be one of the **last** statements of the loop.

For example, if variable *x* is the variable that participates in a loop's Boolean expression, a pre-test loop structure should be in the following form,

```
Initialize x while ( Boolean_Expression(x) )
{
    A statement or block of statements
    Update/alter x
}
```

and a post-test loop structure should be in the following form,

```
Initialize x do {
    A statement or block of statements
    Update/alter x
} while ( Boolean_Expression(x) );
```

where

- ► *Initialize x* is any statement that assigns an initial value to variable *x*. It can be either an input statement such as `cin.nextLine()`, or an assignment statement using the value assignment operator (=). In a post-test loop structure though, this statement may sometimes be redundant and can be omitted since initialization of *x* can occur directly inside the loop.
- ► *Boolean_Expression(x)* can be any Boolean expression from a simple to a complex one, dependent on variable *x*.

- *Update/alter x* is any statement that alters the value of *x*, such as another input statement, an assignment statement using the value assignment operator (=), or even compound assignment operators. It is important that this statement must be positioned just before the point where the loop's Boolean expression is evaluated. This means it should be one of the **last** statements within the loop.

Following are some examples that use the “Ultimate” rule.

Example 1

```
a = Integer.parseInt(cin.nextLine()); //Initialization of a while (a > 0) { //Boolean expression dependent on a System.out.println(a); a = a - 1; //Update/alteration of a }
```

Example 2

```
a = Integer.parseInt(cin.nextLine()); //Initialization of a b = Integer.parseInt(cin.nextLine()); //Initialization of b while (a > b) { //Boolean expression dependent on a and b System.out.println(a + " " + b); a = Integer.parseInt(cin.nextLine()); //Update/alteration of a b = Integer.parseInt(cin.nextLine()); //Update/alteration of b }
```

Example 3

```
s = 0; //Initialization of s do {
   y = Integer.parseInt(cin.nextLine()); s = s + y; //Update/alteration of s } while (s < 1000); //Boolean expression dependent on s
```

Example 4

```
y = 0; //Initialization of y do {
   y = Integer.parseInt(cin.nextLine()); //Update/alteration of y } while (y < 0); //Boolean expression dependent on y
```

In this example, though, initialization of variable y outside the loop is redundant and can be omitted, as shown here.

```
do {
   y = Integer.parseInt(cin.nextLine()); //Initialization and update/alteration of y } while (y < 0); //Boolean expression dependent on y
```

Example 5

```
odd = 0; //Initialization of odd even = 0; //Initialization of even while (odd + even < 5) { //Boolean expression dependent on odd and even x = Integer.parseInt(cin.nextLine()); if (x % 2 == 0) {
      even++; //Update/alteration of even
   }
   else {
      odd++; //Update/alteration of odd
   }
}
System.out.println("Odds: " + odd + " Evens: " + even);
```

Now, you will realize why you should always follow the “Ultimate” rule”! Let's take a look at the following exercise: Write a code fragment that lets the user enter numbers repeatedly until three positive numbers are entered in total.

This exercise was given to a class, and a student gave the following code fragment as an answer.

```
int positivesCount; double x;
positivesCount = 0;
x = Double.parseDouble(cin.nextLine()); while (positivesCount != 3) {
   if (x > 0) {
      positivesCount += 1;
   }
   x = Double.parseDouble(cin.nextLine()); }
System.out.println("Three Positives provided!");
```

At first glance it appears to be correct. It lets the user enter a number, enters the loop, checks whether the user-provided number is positive or not, then lets the user enter a second number, and so on. However, this code contains a logic error —and unfortunately, it's a tricky one. Can you spot it?

Follow the flow of execution by trying various input values—positives, negatives, or even zeros. When the user enters a positive number, the variable `positivesCount` increments by one; and when they enter a negative number or zero, it remains unchanged. Everything appears to runs smoothly, doesn't it? —so smoothly that it might make you question if this book is reliable or if you should throw it away!

The problem becomes evident only when the user attempts to enter all three of the expected positive values. The trace table that follows can help you determine where the problem lies. Let's assume that the user wants to enter the values 5, −10, −2, 4, and 20.

Step	Statement	Notes	positivesCount	x
1	`positivesCount = 0`		**0**	?
2	`x = Double.parseD…`		0	**5.0**
3	`while (positivesCount != 3)`	This evaluates to `true`		
4	`if (x > 0)`	This evaluates to `true`		
5	`positivesCount += 1`		**1**	5.0
6	`x = Double.parseD…`		1	**−10.0**
7	`while (positivesCount != 3)`	This evaluates to `true`		
8	`if (x > 0)`	This evaluates to `false`		
9	`x = Double.parseD…`		1	**−2.0**
10	`while (positivesCount != 3)`	This evaluates to `true`		

11	`if (x > 0)`	This evaluates to `false`		
12	`x = Double.parseD…`		1	**4.0**
13	`while (positivesCount != 3)`	This evaluates to `true`		
14	`if (x > 0)`	This evaluates to `true`		
15	`positivesCount += 1`		**2**	4.0
16	`x = Double.parseD…`		2	**20.0**
17	`while (positivesCount != 3)`	This evaluates to `true`		
18	`if (x > 0)`	This evaluates to `true`		
19	`positivesCount += 1`		**3**	20.0
20	`x = Double.parseD…`		3	**???**

And here is the logic error! At step 20, even though the total number of user-provided positives is three, and you expect the execution to end, unfortunately the user is being asked to enter an additional number! But, you needed a code fragment that lets the user enter three positive numbers, not four, right?

This is why you should always go by the book! Let's see how this code fragment should be written.

Since the Boolean expression of the while-loop is dependent on the variable `positivesCount`, this is the variable that must be initialized outside of the loop. This variable must also be updated/altered within the loop. The statement that does this update/alteration must be the last statement within the loop, as shown in the code fragment (in general form) that follows.

```
positivesCount = 0; //Initialization of positivesCount while (positivesCount != 3) { //This is dependent on positivesCount

A statement or block of statements

   if (x > 0) {
      positivesCount += 1; //Update/alteration of positivesCount
   }
}
```

Now you can add any necessary statements to complete the code. The only statements that you need to add here are the statement that lets the user enter a number (this must be done within the loop), and the statement that displays the last message (this must be done when the loop finishes all of its iterations). So, the final code fragment becomes

int positivesCount; **double** x;

```
positivesCount = 0;
while (positivesCount != 3) {
      x = Double.parseDouble(cin.nextLine()); if (x > 0) {
            positivesCount += 1;
      }
}
System.out.println("Three Positives provided!");
```

29.4 Breaking Out of a Loop

Loops can consume too much CPU time so you have to be very careful when you use them. There are times when you need to break out of, or end, a loop before it completes all of its iterations, usually when a specified condition is met.

Suppose there is a hidden password and you somehow know that it is three characters long, containing only digits. The following for-loop performs 900 iterations in an attempt to find that hidden password using a *brute-force attack*.

```
found = false;
for (i = 100; i <= 999; i++) {
   if (i == hiddenPassword) {
      password = i;
      found = true;
   }
}
if (found == true) {
   System.out.println("Hidden password is: " + password); }
```

✎ *A brute-force attack is the simplest method to gain access to anything that is password protected. An attacker tries combinations of letters, numbers, and symbols with the hope of eventually guessing correctly.*

Now, suppose that the hidden password is 123. As you already know, the for-loop iterates a specified number of times, and in this case, it doesn't care whether the hidden password is actually found or not. Even though the password is found in the 24th iteration, the loop unfortunately continues to iterate until variable `i` reaches the value of 999, thus wasting CPU time.

Someone may argue that 800 – 900 iterations are not a big deal, and they would probably be right. However, in large-scale data processing, every iteration counts. Therefore, you should be very careful when using loop control structures, especially those that iterate too many times. What if the hidden password was ten digits long? This would mean that the for-loop would have to perform 9,000,000,000 iterations!

There are two approaches that can help you make programs like the previous one run faster. The main idea, in both of them, is to break out of the loop when a specified condition is met; in this case when the hidden password is found.

First approach – Using the `break` statement You can break out of a loop before it actually completes all of its iterations by using the `break` statement.

Look at the following Java program. When the hidden password is found, the flow of execution immediately exits (breaks out of) the for-loop.

```
found = false;
for (i = 100; i <= 999; i++) {
   if (i == hiddenPassword) {
      password = i;
      found = true;
      break;
   }
}
if (found) {
   System.out.println("Hidden password is: " + password); }
```

✎ *The statement* `if (found)` *is equivalent to the statement* `if (found == true)`.

Second approach – Using a flag The `break` statement doesn't actually exist in all computer languages; and since this book's intent is to teach you "Algorithmic Thinking" (and not just special statements that only Java supports), let's look at an alternate approach.

In the following Java program, when the hidden password is found, the Boolean expression `found == false` forces the flow of execution to exit the loop.

```
found = false;
i = 100;
while (found == false && i <= 999) {
   if (i == hiddenPassword) {
      password = i;
      found = true;
   }
   i++;
}
if (found) {
   System.out.println("Hidden password is: " + password); }
```

✎ *Consider variable* `found` *as a flag. Initially, the flag is not "raised"* `(found = false)` *. The flow of execution enters the loop, and it keeps iterating as long as the flag remains down* (`while found == false …`). *When something occurs within the loop that raises the flag (assigning* `true` *to the variable* `found`*), the flow of execution exits the loop.*

✎ *The* `while (found == false && i <= 999)` *can alternatively be written as* `while (!found && i <= 999)`.

✎ *The* `i <= 999` *Boolean expression is still necessary in case the hidden password is not found.*

29.5 Cleaning Out Your Loops

As already stated, loops can consume too much CPU time, so you must be very careful and use them sparingly. Although a large number of iterations is sometimes inevitable, there are always things that you can do to make your loops perform better.

The next code fragment calculates the sum of the numbers 1, 2, 3, 4, 5, … 10000.

```
s = 0;
i = 1;
do {
   countOfNumbers = 10000; s = s + i;
   i++;
} while (i <= countOfNumbers);
System.out.println(s);
```

What you should always keep in mind when using loops, especially those that perform many iterations, is to avoid putting any statement inside a loop that serves no purposes in that loop. In the previous example, the statement `countOfNumbers = 10000` is such a statement. Unfortunately, as long as it exists inside the loop, the computer executes it 10000 times for no reason, which of course affects the computer's performance.

To resolve this problem, you can simply move this statement outside the loop, as follows.

```
countOfNumbers = 10000;
s = 0;
i = 1;
do {
   s = s + i;
   i++;
} while (i <= countOfNumbers);
System.out.println(s);
```

Exercise 29.5-1 Cleaning Out the Loop

The following code fragment calculates the average value of numbers 1, 2, 3, 4, … 10000. Try to move as many statements as possible outside the loop to make the program more efficient.

```
s = 0;
for (i = 1; i <= 10000; i++) {

```

```
   s = s + i;
   average = s / 10000; }
System.out.println(average);
```

Solution One very common mistake that novice programmers make when calculating average values is to put the statement that divides the total sum by how many numbers there are in the sum (here `average = s / 10000`) inside the loop. Think about it! Imagine that you want to calculate your average grade in school. Your first step would be to calculate the sum of the grades for all 10 courses that you're taking. Then, when all your grades have been summed up, you would divide that sum by 10. This means that you would perform 10 additions and only one division.

✎ *Calculating an average is a two-step process.*

Therefore, it is pointless to calculate the average value inside the loop. You can move this statement outside and right after the loop, and leave the loop just to sum up the numbers as follows.

```
s = 0;
for (i = 1; i <= 10000; i++) {
   s = s + i;
}
average = s / 10000;
System.out.println(average);
```

Exercise 29.5-2 Cleaning Out the Loop

The next formula

$$S = \frac{1}{1^1 + 2^2 + 3^3 + \cdots + N^N} + \frac{2}{1^1 + 2^2 + 3^3 + \cdots + N^N} + \cdots + \frac{N}{1^1 + 2^2 + 3^3 + \cdots + N^N}$$

is solved using the following code fragment, where N is provided by the user.

```
int n, i, j, denom; double s;
System.out.print("Enter N: "); n = Integer.parseInt(cin.nextLine()); s = 0;
for (i = 1; i <= n; i++) {
   denom = 0;
   for (j = 1; j <= n; j++) {
      denom += Math.pow(j, j);
   }
   s += i / (double)denom; }
System.out.println(s);
```

Try to move as many statements as possible outside the loop to make the code more efficient.

Solution As you can see from the formula, the denominator is common for all fractions. Thus, it is pointless to calculate it again and again for every fraction.

You can calculate the denominator just once and use the result many times, as follows.

```
int n, i, j, denom; double s;
System.out.print("Enter N: "); n = Integer.parseInt(cin.nextLine());
denom = 0; [More…]
for (j = 1; j <= n; j++) {
   denom += Math.pow(j, j); }
s = 0;
for (i = 1; i <= n; i++) {
   s += i / (double)denom; }
System.out.println(s);
```

29.6 Endless Loops and How to Stop Them

All while-loops must include something inside the loop that eventually leads the flow of execution to exit the loop. But mistakes do happen! For instance, the following code fragment contains an endless loop. Unfortunately, the programmer forgot to increase variable `i` inside the loop; therefore, variable `i` can never reach the value 10.

```
i = 1;
while (i != 10) {
   System.out.println("Hello there!"); }
```

✎ *If a loop cannot stop iterating, it is called an endless loop or an infinite loop.*

Another mistake that a programmer can make is something like the following:

```
i = 1;
while (i != 10) {
    System.out.println("Hello there!"); i += 2;
}
```

Even though this code fragment does contain a statement that increases variable `i` inside the loop (`i += 2`), unfortunately the flow of execution never exits the loop because the value 10 is never assigned to the variable `i`.

An endless loop continues to iterate forever, and the only way to stop it from iterating is to use magic forces! For example, when an application in a Windows operating system “hangs” (probably because the flow of execution entered an endless loop), the user must use the key combination ALT+CTRL+DEL to force the application to end.

In Visual Studio Code, when you accidentally write and execute an endless loop, you can simply click on the “Stop” toolbar icon, and the execution will stop.

✎ *In Visual Studio Code, an alternative way to force the application to end is to use the key combination CTRL+C within the Terminal window.*

29.7 The "From Inner to Outer" Method

From inner to outer is a method proposed by this book to help you learn "Algorithmic Thinking" from the inside out. This method first manipulates and designs the inner (nested) control structures and then, as the algorithm (or the program) is developed, more and more control structures are added, nesting the previous ones. This method can be used in large and complicated control structures as it helps you design error-free flowcharts or even Java programs. This book uses this method wherever and whenever it seems necessary.

Let's try the following example.

Write a Java program that displays the following multiplication table as it is shown below.

```
1x1=1   1x2=2    1x3=3    1x4=4    1x5=5    1x6=6    1x7=7    1x8=8    1x9=9    1x10=10
2x1=2   2x2=4    2x3=6    2x4=8    2x5=10   2x6=12   2x7=14   2x8=16   2x9=18   2x10=20
3x1=3   3x2=6    3x3=9    3x4=12   3x5=15   3x6=18   3x7=21   3x8=24   3x9=27   3x10=30
4x1=4   4x2=8    4x3=12   4x4=16   4x5=20   4x6=24   4x7=28   4x8=32   4x9=36   4x10=40
5x1=5   5x2=10   5x3=15   5x4=20   5x5=25   5x6=30   5x7=35   5x8=40   5x9=45   5x10=50
6x1=6   6x2=12   6x3=18   6x4=24   6x5=30   6x6=36   6x7=42   6x8=48   6x9=54   6x10=60
7x1=7   7x2=14   7x3=21   7x4=28   7x5=35   7x6=42   7x7=49   7x8=56   7x9=63   7x10=70
8x1=8   8x2=16   8x3=24   8x4=32   8x5=40   8x6=48   8x7=56   8x8=64   8x9=72   8x10=80
9x1=9   9x2=18   9x3=27   9x4=36   9x5=45   9x6=54   9x7=63   9x8=72   9x9=81   9x10=90
```

According to the "from inner to outer" method, you start by writing the inner control structure, and then, when everything is tested and operates fine, you can add the outer control structure(s).

So, let's try to display only the first line of the multiplication table. If you examine this line, it reveals that, in each multiplication, the multiplicand is always 1. Let's consider the multiplicand as variable `i` with a value of 1. The loop control structure that displays only the first line of the multiplication table is as follows.

Code Fragment 1

```
for (j = 1; j <= 10; j++) {
   System.out.print(i + "x" + j + "=" + i * j + "\t"); }
```

If you execute this code fragment, the result is

```
1x1=1   1x2=2    1x3=3    1x4=4    1x5=5    1x6=6    1x7=7    1x8=8    1x9=9    1x10=10
```

The special sequence of characters `\t` *"displays" a tab character after each iteration. This ensures that everything is aligned properly.*

The inner (nested) loop control structure is ready. What you need now is a way to execute this control structure nine times, but each time variable `i` must contain a different value, from 1 to 9. This can be achieved as follows.

```
Main Code
for (i = 1; i <= 9; i++) {
    Code Fragment 1: Display one single line of
                     the multiplication table
System.out.println(); }
```

✎ *The* `System.out.println()` *statement is used to "display" a line break between lines.*

After embedding **Code Fragment 1** in **Main Code**, the final Java program becomes Project_29.7

```
public static void main(String[] args) throws Exception {
   int i, j;
   for (i = 1; i <= 9; i++) {
      for (j = 1; j <= 10; j++) { [More…]
         System.out.print(i + "x" + j + "=" + i * j + "\t");
      }
      System.out.println();
   }
}
```

29.8 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) When the number of iterations is unknown, you can use a definite loop.
2) When the number of iterations is known, you cannot use a post-test loop structure.
3) According to the "Ultimate" rule, in a pre-test loop structure, the initialization of the variable that participates in the loop's Boolean expression must be done inside the loop.
4) According to the "Ultimate" rule, in a pre-test loop structure, the statement that updates/alters the value of the variable that participates in the loop's Boolean expression must be the last statement within the loop.
5) According to the "Ultimate" rule, in a post-test loop structure, the initialization of the variable that participates in the loop's Boolean

expression can sometimes be done inside the loop.

6) According to the “Ultimate” rule, in a post-test loop structure, the update/alteration of the variable that participates in the loop's Boolean expression must be the first statement within the loop.

7) In Java, you can break out of a loop before it completes all iterations using the `exit` statement.

8) A statement that assigns a constant value to a variable is better placed inside a loop control structure.

9) In the following code fragment, there is at least one statement that can be moved outside the for-loop.

```
for (i = 1; i <= 30; i++) {
   a = "Hello"; System.out.println(a); }
```

10) In the following code fragment, there is at least one statement that can be moved outside the while-loop.

```
s = 0;
count = 1;
while (count < 100) {
   a = Integer.parseInt(cin.nextLine()); s += a;
   average = s / (double)count; count++;
}
System.out.println(average);
```

11) In the following code fragment, there is at least one statement that can be moved outside the while-loop.

```
s = 0;
y = Integer.parseInt(cin.nextLine()); while (y != -99) {
   s = s + y;
   y = Integer.parseInt(cin.nextLine()); }
```

12) The following code fragment satisfies the property of finiteness.

```
i = 1;
while (i != 100) {
   System.out.println("Hello there!"); i += 5;
}
```

13) When the not equal (!=) comparison operator is used in the Boolean expression of a pre-test loop structure, the loop always iterates endlessly.

14) The following code fragment satisfies the property of finiteness.

```
i = 0;
do {
   System.out.println("Hello there!"); i += 5;
} while (i < 100);
```

29.9 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) When the number of iterations is unknown, you can use
   a) the pre-test loop structure.
   b) the post-test loop structure.
   c) all of the above

2) When the number of iterations is known, you can use
   a) the pre-test loop structure.
   b) the post-test loop structure.
   c) a for-loop.
   d) all of the above

3) According to the "Ultimate" rule, in a pre-test loop structure, the initialization of the variable that participates in the loop's Boolean expression must be done
   a) inside the loop.
   b) outside the loop.
   c) all of the above

4) According to the "Ultimate" rule, in a pre-test loop structure, the update/alteration of the variable that participates in the loop's Boolean expression must be done
   a) inside the loop.
   b) outside the loop.
   c) all of the above

5) According to the "Ultimate" rule, in a post-test loop structure, the initialization of the variable that participates in the loop's Boolean expression can be done
   a) inside the loop.
   b) outside the loop.
   c) all of the above

6) In the following code fragment

```
s = 0;
for (i = 1; i <= 100; i++) {
      s = s + i;
      x = 100.0;
      average = s / x; }
```

the number of statements that can be moved outside of the for-loop is
   a) 0.
   b) 1.
   c) 2.
   d) 3.

7) When this comparison operator is used in the Boolean expression of a post-test loop structure, the loop iterates forever.
   a) ==
   b) !=
   c) it depends

29.10 Review Exercises

Complete the following exercises.

1) The following program is supposed to prompt the user to enter names repeatedly until the word “STOP” (used as a name) is entered. At the end, the program must display the total number of names entered as well as how many of these names were not “John”.

```
countNames = 0;
countNotJohns = 0;
name = "";
while (!name.equals("STOP")) {
   System.out.print("Enter a name: "); name = cin.nextLine(); countNames++; if
   (!name.equals("John")) {
      countNotJohns++;
   }
}
System.out.println("Total names entered: " + countNames); System.out.println("Names
other than John entered: " + countNotJohns);
```

However, the program displays wrong results! Using the “Ultimate” Rule, try to modify the program so that it displays the correct results.

2) Write a Java program that prompts the user to enter some text. The text can be either a single word or a whole sentence. Then, the program must display a message stating whether the user-provided text is one single word or a complete sentence.

Hint: Search for a space character! If a space character is found, it means that the user entered a sentence. The program must stop searching further when it finds at least one space character.

3) Write a Java program that prompts the user to enter a sentence. The program must then display the message “The sentence contains a number” if the sentence contains at least one number. The program must stop searching further when it finds at least one digit.

4) Correct the following code fragment so that it does not iterate endlessly.

```
System.out.println("Printing all integers from 1 to 100"); i = 1;
while (i < 101) {
   System.out.println(i); }
```

5) Correct the Boolean expression of the following loop control structure so that it does not iterate endlessly.

```
System.out.println("Printing odd integers from 1 to 99"); i = 1;
while (i != 100)) {
   System.out.println(i); i += 2;
}
```

6) The following code fragment calculates the average value of 100 numbers entered by the user. Try to move as many statements as possible outside the loop to make it more efficient.

```
s = 0;
i = 1;
do {
   count = 100; number = Double.parseDouble(cin.nextLine()); s = s + number; average = s
   / count; i++;
} while (i <= count); System.out.println(average);
```

7) The following formula

$$S = \frac{1}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot 100} + \frac{2}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot 100} + \cdots + \frac{100}{1 \cdot 2 \cdot 3 \cdot \ldots \cdot 100}$$

is solved using the following code fragment

```
int i, j, denom; double s;
s = 0;
for (i = 1; i <= 100; i++) {
      denom = 1;
      for (j = 1; j <= 100; j++) {
            denom *= j;
      }
      s += i / (double)denom; }
System.out.println(s);
```

Try to move as many statements as possible outside the loop to make it more efficient.

8) Write a Java program that displays every combination of two integers as well as their resulting product, for pairs of integers between 1 and 4. The output must display as follows.

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
…
…
```

```
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
```

9) Write a Java program that displays the multiplication table for pairs of integers between 1 and 12, as shown next. Please note that the output is aligned with tabs.

```
    |   1   2   3   4   5   6   7   8   9   10  11  12
------------------------------------------------------
1   |   1   2   3   4   5   6   7   8   9   10  11  12
2   |   2   4   6   8   10  12  14  16  18  20  22  24
3   |   3   6   9   12  15  18  21  24  27  30  33  36
…   |   …   …   …   …   …   …   …   …   …   …   …   …
11  |   11  22  33  44  55  66  77  88  99  110 121 132
12  |   12  24  36  48  60  72  84  96  108 120 132 144
```

10) Write a Java program that prompts the user to enter an integer and then displays the multiplication table for pairs of integers between 1 and that integer. For example, if the user enters the value 5, the output must be as shown next. Please note that the output is aligned with tabs.

```
    |   1   2   3   4   5
------------------------
1   |   1   2   3   4   5
2   |   2   4   6   8   10
3   |   3   6   9   12  15
4   |   4   8   12  16  20
5   |   5   10  15  20  25
```

Chapter 30
More with Loop Control Structures

30.1 Simple Exercises with Loop Control Structures

Exercise 30.1-1 Counting the Numbers According to Which is Greater

Write a Java program that prompts the user to enter 10 pairs of numbers and then counts and displays the number of times that the first user-provided number was greater than the second one and the number of times that the second one was greater than the first one.

Solution

The Java program is as follows. It uses variable `countA` to count the number of times that the first user-provided number was greater than the second one and variable `countB` to count the number of times that the second one was greater than the first one.

Project_30.1-1

```
public static void main(String[] args) throws Exception {
   int countA, countB, i, a, b;
   countA = 0;
   countB = 0;
   for (i = 1; i <= 10; i++) {
      System.out.print("Enter number A: ");
      a = Integer.parseInt(cin.nextLine());
      System.out.print("Enter number B: ");
      b = Integer.parseInt(cin.nextLine());
      if (a > b) {
         countA++;
      }
      else if (b > a) {
         countB++;
      }
   }
   System.out.println(countA + " " + countB); }
```

A reasonable question that someone may ask is "*Why is a multiple-decision control structure being used? Why not use a dual-alternative decision structure instead?*"

Suppose, indeed, that a dual-alternative decision structure, such as the following, is used.

```
if (a > b) {
   countA++;
}
else {
   countB++;
}
```

In this decision control structure, the variable `countB` would increment when variable `b` is greater than variable `a` (this is desirable) but also when variable `b` is equal to variable `a` (this is undesirable). Using a multiple-decision control structure instead would ensure that variable `countB` increments only when variable `b` is greater than (and not when it is equal to) variable `a`.

Exercise 30.1-2 Counting the Numbers According to Their Digits

Write a Java program that prompts the user to enter 20 integers and then counts and displays the total number of one-digit, two-digit, and three-digit integers. Assume that the user enters values between 1 and 999.

Solution

Using knowledge from Exercise 18.1-2, the Java program is as follows.

Project_30.1-2

```
public static void main(String[] args) throws Exception {
   int count1, count2, count3, i, a;
   count1 = count2 = count3 = 0;
   for (i = 1; i <= 20; i++) {
      System.out.print("Enter a number: ");
```

```
      a = Integer.parseInt(cin.nextLine());
      if (a <= 9) {
         count1++;
      }
      else if (a <= 99) {
         count2++;
      }
      else {
         count3++;
      }
   }
   System.out.println(count1 + " " + count2 + " " + count3); }
```

Exercise 30.1-3 How Many Numbers Fit in a Sum

Write a Java program that lets the user enter numeric values repeatedly until the sum of them exceeds 1000. At the end, the program must display the total quantity of numbers entered.

Solution

In this case, since the exact number of iterations is unknown, a definite loop cannot be used; an indefinite loop is required. Let's employ a pre-test loop to create that loop. However, to ensure the program is free of logic errors, it is crucial to adhere to the "Ultimate" rule discussed in Section 29.3. According to this rule, the pre-test loop structure should be as follows, given in general form.

```
Initialize total
while (total <= 1000) {
   A statement or block of statements
   Update/alter total
}
```

Since loop's Boolean expression depends on variable `total`, this is the variable that must be initialized before the loop starts and also updated (altered) within the loop. And more specifically, the statement that updates/alters variable `total` must be the **last** statement of the loop. Following this, the Java program becomes Project_30.1-3

```
public static void main(String[] args) throws Exception {
   int count; double total, x;
   count = 0;
   total = 0; //Initialization of total while (total <= 1000) { //Boolean expression dependent on total
   x = Double.parseDouble(cin.nextLine());
      count++;
      total += x; //Update/alteration of total
   }
   System.out.println(count); }
```

Exercise 30.1-4 Finding the Total Number of Positive Integers

Write a Java program that prompts the user to enter integer values repeatedly until a real one is entered. At the end, the program must display the total number of positive integers entered.

Solution

Once again, you don't know the exact number of iterations, so you cannot use a for-loop.

According to the "Ultimate" rule, the pre-test loop structure should be as follows, given in general form.

```
System.out.print("Enter a number: "); x = Double.parseDouble(cin.nextLine()); //Initialization of x while ( (int)(x) == x ) {
//Boolean expression dependent on x
A statement or block of statements
   System.out.print("Enter a number: "); x = Double.parseDouble(cin.nextLine()); //Update/alteration of x }
```

The final Java program is as follows.

Project_30.1-4

```
public static void main(String[] args) throws Exception {
   double x; int count;
```

```
   count = 0;
   System.out.print("Enter a number: "); x = Double.parseDouble(cin.nextLine()); while ((int)(x) == x) {
      if (x > 0) {
         count++;
      }
      System.out.print("Enter a number: ");
      x = Double.parseDouble(cin.nextLine());
   }
   System.out.println(count); }
```

✎ *Note that the program operates properly even when the first user-provided number is a real (a float); the pre-test loop structure ensures that the flow of execution will never enter the loop for any real numbers!*

Exercise 30.1-5 Iterating as Many Times as the User Wishes

Write a Java program that prompts the user to enter two numbers and then calculates and displays the first number raised to the power of the second one. The program must iterate as many times as the user wishes. At the end of each calculation, the program must prompt the user if they wish to calculate again. If the answer is "yes" the program must repeat; it must end otherwise. Make your program accept the answer in all possible forms such as "yes", "YES", "Yes", or even "YeS".

Solution

According to the "Ultimate" rule, the pre-test loop structure should be as follows, given in general form.

```
answer = "YES"; //Initialization of answer while (answer.toUpperCase().equals("YES")) {

   Prompt the user to enter two numbers and then
   calculate and display the first number raised
   to the power of the second one.

   System.out.print("Would you like to repeat? "); answer = cin.nextLine(); //Update/alteration of answer }
```

✎ *The* `toUpperCase()` *method ensures that the program operates properly for any user-provided answer: "yes", "YES", "Yes", or even "YeS" or "yEs"!*

However, instead of using the pre-test loop structure, let's employ the post-test loop structure this time. This is a better approach, as the initialization of the `answer` variable outside of the loop can be omitted. Unlike the pre-test loop structure, the flow of execution enters the loop in either way, and the initialization of the `answer` will be done inside the post-test loop, as shown in the code fragment (given in general form) that follows.

```
do {

   Prompt the user to enter two numbers and then
   calculate and display the first number raised
   to the power of the second one.

   System.out.print("Would you like to repeat? "); answer = cin.nextLine(); //Initialization and update/alteration of answer }
   while (answer.toUpperCase().equals("YES"));
```

The solution to this exercise becomes Project_30.1-5

```
public static void main(String[] args) throws Exception {
   int a, b; double result; String answer;
   do {
      System.out.println("Enter two numbers: ");
      a = Integer.parseInt(cin.nextLine());
      b = Integer.parseInt(cin.nextLine());
      result = Math.pow(a, b);
      System.out.println("The result is: " + result);
      System.out.print("Would you like to repeat? ");
      answer = cin.nextLine();
   } while (answer.toUpperCase().equals("YES")); }
```

Exercise 30.1-6 Finding the Sum of the Digits

Write a Java program that lets the user enter an integer and then calculates the sum of its digits.

Solution

In Exercise 13.1-2, you learned how to split the digits of an integer when its total number of digits was known. In this exercise however, the user is allowed to enter any value, no matter how small or large. Thus, the total number of the digits is an unknown quantity.

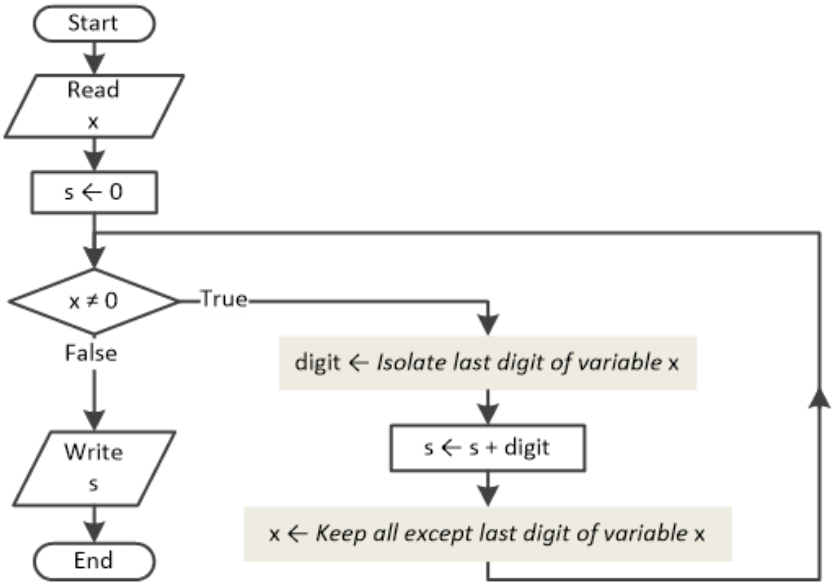
To solve this exercise, a loop control structure could be used. However, there are two approaches that you can use.

First approach

In this approach, the main idea is to isolate one digit at each iteration. However, the challenge lies in determining the total number of iterations required, as it hinges on the size of the user-provided integer. So, does this pose a roadblock? Certainly not!

Within the loop, the user-provided integer should undergo a continuous reduction with each iteration until it eventually reaches zero. That value of zero can act as a condition to stop the loop control structure from iterating. For instance, if the user-provided number is 4753, it should become 475 in the first iteration, 47 in the second iteration, then 4, and ultimately 0. Once it reaches 0, the iterations must stop.

Let's try to comprehend the proposed solution using the following flowchart. Some statements are written in general form.



The statement

`digit ← Isolate last digit of variable x.`

can be written using the well-known MOD 10 operation as shown here.

```
digit ← x MOD 10
```

The whole concept, however, relies on the statement `x ← Keep all except last digit of variable x`.

This is the statement that eventually zeros the value of variable `x`, and the flow of execution then exits the loop. To write this statement you can use a DIV 10 operation as shown here.

```
x ← x DIV 10
```

Accordingly, the Java program becomes Project_30.1-6a

```
public static void main(String[] args) throws Exception {
    int x, s, digit;
    x = Integer.parseInt(cin.nextLine());
    s = 0;
    while (x != 0) {
        digit = x % 10; //This is the x MOD 10 operation
        s = s + digit;
        x = (int)(x / 10); //This is the x DIV 10 operation
    }
```

System.out.**println**(s); }

Let's create a trace table for the input value 4753 to better understand what is really happening.

Step	Statement	Notes	x	digit	s
1	`x = Integer.parseI…`	User enters the value 4753	**4753**	?	?
2	`s = 0`		4753	?	**0**
3	`while (x != 0)`	This evaluates to `true`			
4	`digit = x % 10`		4753	**3**	0
5	`s = s + digit`		4753	3	**3**
6	`x = (int)(x / 10)`		**475**	3	3
7	`while (x != 0)`	This evaluates to `true`			
8	`digit = x % 10`		475	**5**	3
9	`s = s + digit`		475	5	**8**
10	`x = (int)(x / 10)`		**47**	5	8
11	`while (x != 0)`	This evaluates to `true`			
12	`digit = x % 10`		47	**7**	8
13	`s = s + digit`		47	7	**15**
14	`x = (int)(x / 10)`		**4**	7	15
15	`while (x != 0)`	This evaluates to `true`			
16	`digit = x % 10`		4	**4**	15
17	`s = s + digit`		4	4	**19**
18	`x = (int)(x / 10)`		**0**	4	19
19	`while (x != 0)`	This evaluates to `false`			
20	`.println(s)`	It displays: 19			

In Java, the result of the division of two integers is always an integer. Thus, in the statement `x = (int)(x / 10)`*, since both variable* `x` *and constant value 10 are integers, the* `(int)` *casting operator is redundant. However, it is a good practice to keep it there just for improved readability.*

Second approach

In this approach, the main idea is to convert the user-provided integer to a string and then use a for-loop to iterate for all its characters (digits). In the for-loop, however, you need to convert each digit from type `char` back to type `int` before it is accumulated in variable `s`. The Java program is as follows.

```
Project_30.1-6b
public static void main(String[] args) throws Exception {
int i, x, s; String xStr, digit;
x = Integer.parseInt(cin.nextLine()); xStr = "" + x; //Convert user's input to
string
s = 0;
for (i = 0; i <= xStr.length() - 1; i++) {
digit = "" + xStr.charAt(i); //Get the "digit" as string
s = s + Integer.parseInt(digit);
}
System.out.println(s); }
```

30.2 Exercises with Nested Loop Control Structures

Exercise 30.2-1 Displaying all Three-Digit Integers that Contain a Given Digit

Write a Java program that prompts the user to enter a digit (0 to 9) and then displays all three-digit integers that contain that user-provided digit at least once. For example, for the user-provided value 7, the values 357, 771, and 700 are such integers.

Solution

There are three different approaches! The first one uses just one for-loop, the second one uses three for-loops, nested one within the other, and the last one converts all three-digit integers to strings. Let's analyze them all!

First approach – Using a for-loop and a decision control structure The main idea is to use a for-loop where the value of variable *counter* goes from 100 to 999. Inside the loop, the *counter* variable is split into its individual digits ($digit_3$, $digit_2$, $digit_1$) and a decision control structure is used to check if at least one of its digits is equal to the provided one. The Java program is as follows.

```
                        Project_30.2-1a
        public static void main(String[] args) throws Exception {
                    int x, i, digit3, r, digit2, digit1;
System.out.print("Enter a digit 0 - 9: "); x = Integer.parseInt(cin.nextLine());
                    for (i = 100; i <= 999; i++) {
                        digit3 = (int)(i / 100);
                              r = i % 100;
                         digit2 = (int)(r / 10);
                            digit1 = r % 10;
                if (digit3 == x || digit2 == x || digit1 == x) {
                              System.out.println(i);
                                      }
                                    }
                                  }
```

Second approach – Using nested loop control structures and a decision control structure The main idea here is to use three for-loops, nested one within the other. In this case, there are three *counter* variables (`digit3`, `digit2`, and `digit1`) and each one of them corresponds to one digit of the three-digit integer. The Java program is as follows.

```
                        Project_30.2-1b
        public static void main(String[] args) throws Exception {
                    int x, digit3, digit2, digit1;
System.out.print("Enter a digit 0 - 9: "); x = Integer.parseInt(cin.nextLine());
                  for (digit3 = 1; digit3 <= 9; digit3++) {
                    for (digit2 = 0; digit2 <= 9; digit2++) {
                      for (digit1 = 0; digit1 <= 9; digit1++) {
                    if (digit3 == x || digit2 == x || digit1 == x) {
                  System.out.println(digit3 * 100 + digit2 * 10 + digit1);
                                        }
                                      }
                                    }
                                  }
                                }
```

If you follow the flow of execution, the value 100 is the first "integer" evaluated (`digit3 = 1`, `digit2 = 0`, `digit1 = 0`). Then, the most-nested loop control structure increments variable `digit1` by one and the next value evaluated is "integer" 101. This continues until `digit1` reaches the value 9; that is, until the "integer" reaches the value 109. The flow of execution then exits the most-nested loop control structure, variable `digit2` increments by one, and the most-nested loop control structure starts over again, thus the values evaluated are the "integers" 110, 111, 112, … 119. The process goes on until all integers up to the value 999 are evaluated.

✎ *Note that variable* `digit3` *starts from 1, whereas variables* `digit2` *and* `digit1` *start from 0. This is necessary since the scale for three-digit numbers begins from 100 and not from 000.*

✎ *Note how the* `System.out.println()` *statement composes the three-digit integer.*

Third approach – Convert all three-digit integers to strings Using a for-loop, the value of variable *counter* goes from 100 to 999. Inside the loop, the *counter* variable is converted to string and the `indexOf()` method checks if the user-provided "digit" exists in the string. The Java program is as follows.

```
Project_30.2-1c
public static void main(String[] args) throws Exception {
                int i; String x;
    System.out.print("Enter a digit 0 - 9: "); x = cin.nextLine();
                for (i = 100; i <= 999; i++) {
if (String.valueOf(i).indexOf(x) != -1) { //Or you can do the following:
                    //if (("" + i).indexOf(x) != -1)
                     System.out.println(i);
                            }
                          }
                        }
```

✎ *Note that variable* `x` *is of type* `String`.

Exercise 30.2-2 Displaying all Instances of a Specified Condition

Write a Java program that displays all three-digit integers in which the first digit is smaller than the second digit and the second digit is smaller than the third digit. For example, the values 357, 456, and 159 are such integers.

Solution

Using knowledge from the previous exercise (Exercise 30.2-1), there are three different approaches! Let's analyze them all!

First approach – Using a for-loop and a decision control structure Using a for-loop and a decision control structure, the Java program is as follows.

```
Project_30.2-2a
public static void main(String[] args) throws Exception {
            int i, r, digit1, digit2, digit3;
              for (i = 100; i <= 999; i++) {
                  digit3 = (int)(i / 100);
                        r = i % 100;
                  digit2 = (int)(r / 10);
                     digit1 = r % 10;
          if (digit3 < digit2 && digit2 < digit1) {
                     System.out.println(i);
                            }
                          }
                        }
```

Second approach – Using nested loop control structures and a decision control structure Using nested loop control structures and a decision control structure, the Java program is as follows.

```
Project_30.2-2b
public static void main(String[] args) throws Exception {
                int digit1, digit2, digit3;
          for (digit3 = 1; digit3 <= 9; digit3++) {
            for (digit2 = 0; digit2 <= 9; digit2++) {
              for (digit1 = 0; digit1 <= 9; digit1++) {
                if (digit3 < digit2 && digit2 < digit1) {
        System.out.println(digit3 * 100 + digit2 * 10 + digit1);
```

```
                }
              }
            }
          }
        }
```

Third approach – Using nested loop control structures only This approach is based on the second approach. The main difference between them is that in this case, variable `digit1` always begins from a value greater than `digit2`, and variable `digit2` always begins from a value greater than `digit3`. In that way, the first integer that will be displayed is 123.

> ✎ *There are no integers below the value 123 and above the value 789 that can validate the Boolean expression* `digit3 < digit2 && digit2 < digit1` *to* `true`.

The Java program is as follows.

```
                    Project_30.2-2c
public static void main(String[] args) throws Exception {
                int digit3, digit2, digit1;
          for (digit3 = 1; digit3 <= 7; digit3++) {
        for (digit2 = digit3 + 1; digit2 <= 8; digit2++) {
          for (digit1 = digit2 + 1; digit1 <= 9; digit1++) {
        System.out.println(digit3 * 100 + digit2 * 10 + digit1);
                            }
                          }
                        }
                      }
```

> ✎ *This solution is the most efficient since it doesn't use any decision control structure and, moreover, the number of iterations is kept to a minimum!*
>
> *As you can see, one problem can have many solutions. It is up to you to find the optimal one!*

30.3 Data Validation with Loop Control Structures

As you already know, data validation is the process of restricting data input, which forces the user to enter only valid values. You have already encountered one method of data validation using decision control structures. Let's recall an example.

```
System.out.print("Enter a non-negative number: "); x = Double.parseDouble(cin.nextLine());
if (x < 0) {
   System.out.print("Error: Negative number entered!"); }
else {
   System.out.println(Math.sqrt(x)); }
```

This approach, however, may not be the most convenient for the user. If they enter an invalid number, the program displays the error message, and the flow of execution inevitably reaches the end. The user must then restart the program to re-enter a valid number.

Next, you will find three approaches given in general form for validating data input using loop control structures. In cases where a user enters an invalid value, the primary objective is to prompt them repeatedly until they eventually provide a valid one. Of course, if the user initially enters a valid value, the flow of execution simply proceeds to the next section of the program.

Which approach you use depends on whether or not you wish to display an error message and whether you wish to display different error messages, one for each type of input error, or just a generic error message for any kind of error.

First approach – Validating data input without error messages To validate data input without displaying any error messages, you can use the following code fragment given in general form.

```
do {
   System.out.print("Prompt message"); input_data = cin.nextLine(); } while (input_data test 1 fails || input_data test 2 fails ||
   …);
```

Second approach – Validating data input with a generic error message To validate data input and display a generic error message (that is, the same error message for any type of input error), you can use the following code fragment given in general form.

```
System.out.print("Prompt message"); input_data = cin.nextLine(); while (input_data test 1 fails || input_data test 2 fails || … )
{
   System.out.println("Error message"); System.out.print("Prompt message"); input_data = cin.nextLine(); }
```

Third approach – Validating data input with different error messages To validate data input and display a different error message for each type of input error, you can use the following code fragment given in general form.

```
do {
   System.out.print("Prompt message"); input_data = cin.nextLine(); failure = false;
   if (input_data test 1 fails) {
      System.out.println("Error message 1");
      failure = true;
   }
   else if (input_data test 2 fails) {
      System.out.println("Error message 2");
      failure = true;
   }
   else if (…
   …
   }
} while (failure);
```

✎ *The statement* `while (failure)` *is equivalent to the statement* `while (failure == true)`.

Exercise 30.3-1 Finding Odd and Even Numbers

Write a Java program that prompts the user to enter a non-negative integer, and then displays a message indicating whether this number is even; it must display "Odd" otherwise. Using a loop control structure, the program must also validate data input, allowing the user to enter only a non-negative integer.

Solution

All three approaches for validating data input that you learned in Section 30.3 will be presented here. But first, let's solve this exercise without data validation.

```
System.out.print("Enter a non-negative integer: "); [More…]
x = Integer.parseInt(cin.nextLine());

if (x % 2 == 0) {
   System.out.println("Even"); }
else {
   System.out.println("Odd"); }
```

Validation Without Error Messages To validate data input without displaying any error messages, use the first approach from Section 30.3. Simply replace the statements marked with a dashed rectangle with the following code fragment.

```
do {
   System.out.print("Enter a non-negative integer: "); x = Double.parseDouble(cin.nextLine()); } while (x < 0 || (int)x != x);
```

The final Java program becomes Project_30.3-1a

```
public static void main(String[] args) throws Exception {
    double x;
    do { [More…]
         System.out.print("Enter a non-negative integer: ");
         x = Double.parseDouble(cin.nextLine());
    } while (x < 0 || (int)x != x);
    if (x % 2 == 0) {
         System.out.println("Even");
    }
    else {
         System.out.println("Odd");
```

```
    }
}
```

✎ *Variable* `x` *is declared as* `double`*. This is necessary in order to allow the user to enter either an integer or a float.*

Validation with a Generic Error Message To validate data input and display a generic error message, replace the statements marked with the dashed rectangle with a code fragment based on the second approach from Section 30.3. The Java program is as follows.

```
Project_30.3-1b
public static void main(String[] args) throws Exception {
                        double x;
    System.out.print("Enter a non-negative integer: ");  [More…]
x = Double.parseDouble(cin.nextLine()); while (x < 0 || (int)x != x) {
   System.out.println("Error! A negative value or a float entered.");
          System.out.print("Enter a non-negative integer: ");
                x = Double.parseDouble(cin.nextLine());
                                }
                        if (x % 2 == 0) {
                    System.out.println("Even");
                                }
                            else {
                    System.out.println("Odd");
                                }
                            }
```

Validation with Different Error Messages Here, the replacing code fragment is based on the third approach from Section 30.3. To validate data input and display a different error message for each type of input error, the Java program is as follows.

```
Project_30.3-1c
public static void main(String[] args) throws Exception {
                    double x; boolean failure;
                        do {  [More…]
          System.out.print("Enter a non-negative integer: ");
                x = Double.parseDouble(cin.nextLine());
                            failure = false;
                              if (x < 0) {
      System.out.println("Error! You entered a negative value");
                            failure = true;
                                }
                      else if ((int)x != x) {
         System.out.println("Error! You entered a float");
                            failure = true;
                                }
                    } while (failure);
                        if (x % 2 == 0) {
                    System.out.println("Even");
                                }
                            else {
                    System.out.println("Odd");
                                }
                            }
```

Exercise 30.3-2 Finding the Sum of Four Numbers

Write a Java program that prompts the user to enter four positive numbers and then calculates and displays their sum. Using a loop control structure, the program must also validate data input and display an error message when the user enters any non-positive value.

Solution

This exercise was already discussed in Exercise 26.1-4. The only difference here is that this program must validate data input and display an error message when the user enters invalid values. For your convenience, the solution proposed in that exercise is reproduced here.

```
total = 0;
for (i = 1; i <= 4; i++) {
   System.out.print("Enter a number: "); [More…]
   x = Double.parseDouble(cin.nextLine());
   total += x;
}
System.out.println(total);
```

The primary purpose of this exercise is to demonstrate how to nest the loop control structure that validates data input into other pre-existing loop control structures. In this exercise, you should replace the statements marked with a dashed rectangle with the following code fragment

System.out.**print**("*Enter a number:* "); x = Double.**parseDouble**(cin.**nextLine**()); **while** (x <= 0) { System.out.**println**("*Please enter a positive value!*"); System.out.**print**("*Enter a number:* "); x = Double.**parseDouble**(cin.**nextLine**()); }

and the final Java program becomes Project_30.3-2

```
public static void main(String[] args) throws Exception {
   int i; double total, x;
   total = 0;
   for (i = 1; i <= 4; i++) {
      System.out.print("Enter a number: "); [More…]
      x = Double.parseDouble(cin.nextLine());
      while (x <= 0) {
         System.out.println("Please enter a positive value!");
         System.out.print("Enter a number: ");
         x = Double.parseDouble(cin.nextLine());
      }
      total += x;
   }
   System.out.println(total); }
```

Note that the replacing code fragment is entirely nested within this outer for-loop.

30.4 Finding Minimum and Maximum Values with Loop Control Structures

In Section 23.2 you learned how to find the minimum and maximum values among four values using single-alternative decision structures. Now, the following code fragment achieves the same result but uses only one variable `w`, for the user-provided values.

```
w = Integer.parseInt(cin.nextLine()); //User enters 1st value maximum = w;
w = Integer.parseInt(cin.nextLine()); //User enters 2nd value if (w > maximum) {
   maximum = w;
}
w = Integer.parseInt(cin.nextLine()); //User enters 3rd value if (w > maximum) {
   maximum = w;
}
w = Integer.parseInt(cin.nextLine()); //User enters 4th value if (w > maximum) {
   maximum = w;
}
```

Except for the first pair of statements, all other blocks of statements are identical. Therefore, you can retain only one of these pairs and enclose it within a loop control structure that performs three iterations, as presented below.

```
w = Integer.parseInt(cin.nextLine()); //User enters 1st value maximum = w;
for (i = 1; i <= 3; i++) {
   w = Integer.parseInt(cin.nextLine()); //User enters 2nd, 3rd and 4th value if (w > maximum) {
```

```
        maximum = w;
    }
}
```

Of course, if you want to allow the user to enter more values, you can simply increase the *final_value* of the for-loop.

Accordingly, a program that finds and displays the heaviest person among 10 individuals is presented next.

```
Project_30.4a
public static void main(String[] args) throws Exception {
                    int w, maximum, i;
   System.out.print("Enter a weight (in pounds): "); w =
      Integer.parseInt(cin.nextLine()); maximum = w;
               for (i = 1; i <= 9; i++) {
      System.out.print("Enter a weight (in pounds): ");
         w = Integer.parseInt(cin.nextLine());
               if (w > maximum) {
                    maximum = w;
                        }
                    }
          System.out.println(maximum); }
```

✎ *Note that the for-loop iterates one time less than the total number of user-provided values.*

Even though this Java program operates fine, let's do something slightly different. Instead of prompting the user to enter the first value before the loop and the remaining nine values within the loop, let's prompt them to enter all values within the loop.

However, the issue that arises here is that, no matter what, an initial value must always be assigned to the variable `maximum` before the loop starts iterating. But, this value cannot be arbitrarily chosen; it depends on the given problem. Therefore, choosing an “almost arbitrary” initial value requires careful consideration, as an incorrect choice may yield inaccurate results.

In this exercise, all user-provided values have to do with people's weight. Since there is no chance of finding any person with a negative weight (at least not on planet Earth), you can safely assign the initial value −1 to variable `maximum`, as follows.

```
Project_30.4b
public static void main(String[] args) throws Exception {
                    int maximum, i, w;
                       maximum = -1;
                  for (i = 1; i <= 10; i++) {
      System.out.print("Enter a weight (in pounds): ");
         w = Integer.parseInt(cin.nextLine());
               if (w > maximum) {
                    maximum = w;
                        }
                    }
          System.out.println(maximum); }
```

Once the flow of execution enters the loop, the user enters the first value and the decision control structure evaluates to `true`. The initial value −1 in variable `maximum` is then overwritten by this first user-provided value and afterward, the flow of execution proceeds normally.

✎ *Note that this method may not be applicable in all cases. If an exercise requires prompting the user to enter any number (not limited to positive ones), this method cannot be applied, as the user could potentially enter only negative values. If this were to occur, the initial value of −1 would never be replaced by any of the user-provided values. This method can be used to find the maximum value only when the lower limit of user-provided values is known, or to find the minimum value only when the upper limit of user-provided values is known. For instance, if the exercise requires finding the lightest person, you can assign the initial value +1500 to variable* `minimum`, *as*

there is no human on Earth who can weigh that much! For reference, Jon Brower Minnoch was an American who, at his peak weight, was recorded as the heaviest human being ever, weighing approximately 1,400 lb!!!!!

Exercise 30.4-1 Validating and Finding the Minimum and the Maximum Value

Write a Java program that prompts the user to enter the weight of 10 people and then finds the lightest and the heaviest weights. Using a loop control structure, the program must also validate data input and display an error message when the user enters any non-positive value, or any value greater than 1500.

Solution

Using the previous exercise as a guide, you should now be able to do this with your eyes closed!

To validate data input, all you have to do is replace the following two lines of code of the previous exercise,

System.out.**print**("*Enter a weight (in pounds):* "); w = Integer.**parseInt**(cin.**nextLine**());

with the following code fragment:

System.out.**print**("*Enter a weight (in pounds):* "); w = Integer.**parseInt**(cin.**nextLine**()); **while** (w < 1 || w > 1500) {

System.out.**println**("*Invalid value! Enter a weight between 1 and 1500 (in pounds):*"); w = Integer.**parseInt**(cin.**nextLine**()); }

Following is the final program that finds the lightest and the heaviest weights.

Project_30.4-1

```
public static void main(String[] args) throws Exception {
   int minimum, maximum, i, w;
   minimum = 1500;
   maximum = 0;
   for (i = 1; i <= 10; i++) {
      System.out.print("Enter a weight (in pounds): "); [More…]
      w = Integer.parseInt(cin.nextLine());
      while (w < 1 || w > 1500) {
         System.out.print("Invalid value! Enter a weight between 1 and 1500 (in pounds): ");
         w = Integer.parseInt(cin.nextLine());
      }
      if (w < minimum) {
         minimum = w;
      }
      if (w > maximum) {
         maximum = w;
      }
   }
   System.out.println(minimum + " " + maximum); }
```

Exercise 30.4-2 Validating and Finding the Hottest Planet

Write a Java program that prompts the user to repeatedly enter the names and the average temperatures of planets from space, until the word "STOP" (used as a name) is entered. In the end, the program must display the name of the hottest planet. Moreover, since −459.67° (on the Fahrenheit scale) is the lowest temperature possible (it is called absolute zero), the program must also validate data input (using a loop control structure) and display an error message when the user enters temperature values lower than absolute zero.

Solution

First, let's write the Java program without using data validation. According to the "Ultimate" rule, the pre-test loop structure should be as follows, given in general form:

System.out.**print**("*Enter the name of a planet:* "); name = cin.**nextLine**(); //*Initialization of name* **while** (!name.**toUpperCase**().**equals**("*STOP*")) {

```
A statement or block of statements
```

System.out.**print**("*Enter the name of a planet:* "); name = cin.**nextLine**(); //*Update/alteration of name* }

Now, let's add the rest of the statements, still without data input validation. Keep in mind that, since value −459.67° is the lower limit of the temperature scale, you can use a value lower than this as the initial value of

variable `maximum`.

```
maximum = -460;
mName = "";
System.out.print("Enter the name of a planet: "); name = cin.nextLine(); //Initialization of name while
(!name.toUpperCase().equals("STOP")) {
   System.out.print("Enter its average temperature: "); t = Double.parseDouble(cin.nextLine());
   if (t > maximum) {
      maximum = t;
      mName = name;
   }
   System.out.print("Enter the name of a planet: "); name = cin.nextLine(); //Update/alteration of name }
if (maximum != -460) {
   System.out.println("The hottest planet is: " + mName); }
else {
   System.out.println("Nothing Entered!"); }
```

✎ *The* `if (maximum != -460)` *statement is required because there is a possibility that the user could enter the word "STOP" right from the beginning.*

To validate the data input, all you have to do is replace the following two lines of code:

System.out.**print**("*Enter its average temperature:* "); t = Double.**parseDouble**(cin.**nextLine**());

with the following code fragment:

System.out.**print**("*Enter its average temperature:* "); t = Double.**parseDouble**(cin.**nextLine**()); **while** (t < -459.67) {

System.out.**print**("*Invalid value! Enter its average temperature:* "); t = Double.**parseDouble**(cin.**nextLine**()); }

The final program is as follows.

🗀 **Project_30.4-2**

```
public static void main(String[] args) throws Exception {
   double maximum, t; String mName, name;
   maximum = -460;
   mName = "";
   System.out.print("Enter the name of a planet: "); name = cin.nextLine(); while (!name.toUpperCase().equals("STOP")) {
      System.out.print("Enter its average temperature: ");
      t = Double.parseDouble(cin.nextLine());
      while (t < -459.67) {
         System.out.print("Invalid value! Enter its average temperature: ");
         t = Double.parseDouble(cin.nextLine());
      }
      if (t > maximum) {
         maximum = t;
         mName = name;
      }
      System.out.print("Enter the name of a planet: ");
      name = cin.nextLine();
   }
   if (maximum != -460) {
      System.out.println("The hottest planet is: " + mName);
   }
   else {
      System.out.println("Nothing Entered!");
   }
}
```

Exercise 30.4-3 "Making the Grade"

In a classroom, there are 20 students. Write a Java program that prompts the teacher to enter the grades (0 - 100) that students received in a math test and then displays the highest grade as well as the number of students that got an "A" (that is, 90 to 100). Moreover, the program must validate data input. User-provided values must be within the range 0 to 100.

Solution

Let's first write the program without data validation. Since the number of students is known, you can use a for-loop. For an initial value of variable `maximum`, you can use value −1 as there is no grade lower than 0.

```
maximum = -1;
count = 0;
for (i = 1; i <= 20; i++) {
   System.out.print("Grade for student No " + i + ": "); grade = Integer.parseInt(cin.nextLine());
   if (grade > maximum) {
      maximum = grade;
   }
   if (grade >= 90) {
      count++;
   }
}
System.out.println(maximum + " " + count);
```

Now, you can deal with data validation. As the wording of the exercise implies, there is no need to display any error messages. So, all you need to do is replace the following two lines of code:

System.out.**print**("*Enter a grade for student No* " + i + "*:* "); grade = Integer.**parseInt**(cin.**nextLine**());

with the following code fragment:

do {

System.out.**print**("*Grade for student No* " + i + "*:* "); grade = Integer.**parseInt**(cin.**nextLine**()); } **while** (grade < 0 || grade > 100);

and the final program becomes Project_30.4-3

```
public static void main(String[] args) throws Exception {
   int maximum, count, i, grade;
   maximum = -1;
   count = 0;
   for (i = 1; i <= 20; i++) {
      do {
         System.out.print("Grade for student No " + i + ": ");
         grade = Integer.parseInt(cin.nextLine());
      } while (grade < 0 || grade > 100);
      if (grade > maximum) {
         maximum = grade;
      }
      if (grade >= 90) {
         count++;
      }
   }
   System.out.println(maximum + " " + count); }
```

30.5 Using Loop Control Structures to Solve Mathematical Problems

Exercise 30.5-1 Calculating the Area of as Many Triangles as the User Wishes

Write a Java program that prompts the user to enter the lengths of all three sides A, B, and C of a triangle and then calculates and displays its area. You can use Heron's formula, $Area = \sqrt{S(S-A)(S-B)(S-C)}$

where S is the semi-perimeter $S = \frac{A+B+C}{2}$

The program must iterate as many times as the user wishes. At the end of each area calculation, the program must ask the user if they wish to calculate the area of another triangle. If the answer is "yes" the program must repeat; it must end otherwise. Make your program accept the answer in all possible forms such as "yes", "YES", "Yes", or even "YeS".

Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any non-positive value.

Solution

According to the "Ultimate" rule, the post-test loop structure should be as follows, given in general form.

```
answer = "yes"; //Initialization of answer (redundant).
do {
```

```
    Prompt the user to enter the lengths of all
    three sides A, B, C of a triangle and then
    calculate and display its area.
    System.out.print("Would you like to repeat? "); answer = cin.nextLine(); //Update/alteration of answer
} while (answer.toUpperCase().equals("YES"));
```

The `ToUpperCase()` *method ensures that the program operates properly for any user-provided answer "Yes", "yes", "YES" or even "YeS" or "yEs"!*

The solution to this exercise is as follows.

Project_30.5-1

```
public static void main(String[] args) throws Exception {
   double a, b, c, s, area; String answer;
   do {
      //Prompt the user to enter the length of side A
      System.out.print("Enter side A: ");
      a = Double.parseDouble(cin.nextLine());
      while (a <= 0) {
         System.out.print("Invalid side. Enter side A: ");
         a = Double.parseDouble(cin.nextLine());
      }
      //Prompt the user to enter the length of side B
      System.out.print("Enter side B: ");
      b = Double.parseDouble(cin.nextLine());
      while (b <= 0) {
         System.out.print("Invalid side. Enter side B: ");
         b = Double.parseDouble(cin.nextLine());
      }
      //Prompt the user to enter the length of side C
      System.out.print("Enter side C: ");
      c = Double.parseDouble(cin.nextLine());
      while (c <= 0) {
         System.out.print("Invalid side. Enter side C: ");
         c = Double.parseDouble(cin.nextLine());
      }
      //Calculate and display the area of the triangle
      s = (a + b + c) / 2;
      area = Math.sqrt(s * (s - a) * (s - b) * (s - c));
      System.out.println("The area is: " + area);
      System.out.print("Would you like to repeat? ");
      answer = cin.nextLine();
   } while (answer.toUpperCase().equals("YES")); }
```

Exercise 30.5-2 Finding x and y

Write a Java program that displays all possible integer values of x and y within the range −20 to +20 that validate the following formula: $3x^2 - 6y^2 = 6$

Solution

If you just want to display **all** possible combinations of variables x and y, you can use the following code fragment.

```
int x, y;
for (x = -20; x <= 20; x++) {
   for (y = -20; y <= 20; y++) {
      System.out.println(x + " " + y);
   }
}
```

However, from all those combinations, you need only those that validate the expression $3x^2 - 6y^2 = 6$. A decision control structure is perfect for that purpose! The final Java program is as follows.

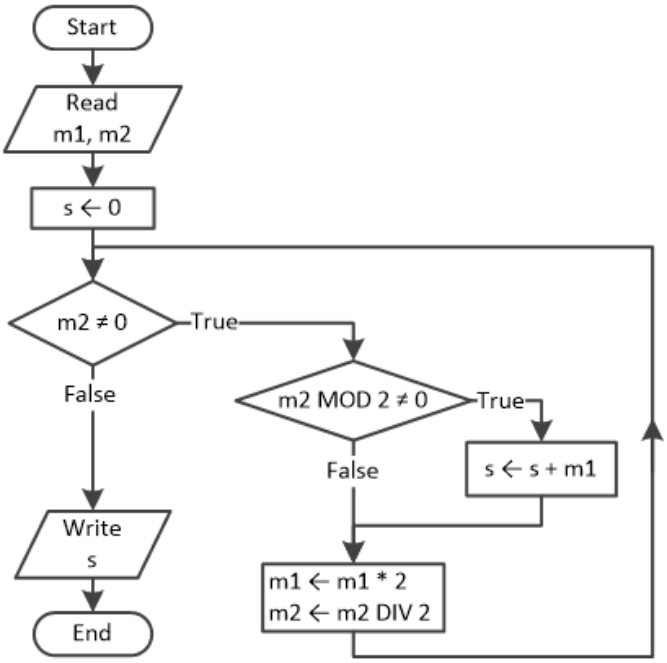
Project_30.5-2

```
public static void main(String[] args) throws Exception {
   int x, y;
   for (x = -20; x <= 20; x++) {
```

```
      for (y = -20; y <= 20; y++) {
         if (3 * Math.pow(x, 2) - 6 * Math.pow(y, 2) == 6) {
            System.out.println(x + " " + y);
         }
      }
   }
}
```

Exercise 30.5-3 The Russian Multiplication Algorithm

You can multiply two positive integers using the "Russian multiplication algorithm", which is presented in the following flowchart.



Write the corresponding Java program and create a trace table to determine the values of the variables in each step for the input values 5 and 13.

Solution

In the given flowchart, a single-alternative decision structure is nested within a pre-test loop structure. The corresponding Java program is as follows.

Project_30.5-3

```
public static void main(String[] args) throws Exception {
   int m1, m2, s;
   m1 = Integer.parseInt(cin.nextLine()); m2 = Integer.parseInt(cin.nextLine());
   s = 0;
   while (m2 != 0) {
      if (m2 % 2 != 0) {
         s += m1;
      }
      m1 *= 2;
      m2 = (int)(m2 / 2);
   }
   System.out.println(s); }
```

For the input values of 5 and 13, the trace table looks like this.

Step	Statement	Notes	m1	m2	s
1	`m1 = Integer.parseI…`	User enters the value 5	**5**	?	?
2	`m2 =`	User enters the value 13	5	**13**	?

	`Integer.parseI…`				
3	`s = 0`		5	13	**0**
4	`while (m2 != 0)`	This evaluates to `true`			
5	`if (m2 % 2 != 0)`	This evaluates to `true`			
6	`s += m1`		5	13	**5**
7	`m1 *= 2`		**10**	13	5
8	`m2 = (int)(m2 / 2)`		10	**6**	5
9	`while (m2 != 0)`	This evaluates to `true`			
10	`if (m2 % 2 != 0)`	This evaluates to `false`			
11	`m1 *= 2`		**20**	6	5
12	`m2 = (int)(m2 / 2)`		20	**3**	5
13	`while (m2 != 0)`	This evaluates to `true`			
14	`if (m2 % 2 != 0)`	This evaluates to `true`			
15	`s += m1`		20	3	**25**
16	`m1 *= 2`		**40**	3	25
17	`m2 = (int)(m2 / 2)`		40	**1**	25
18	`while (m2 != 0)`	This evaluates to `true`			
19	`if (m2 % 2 != 0)`	This evaluates to `true`			
20	`s += m1`		40	1	**65**
21	`m1 *= 2`		**80**	1	65
22	`m2 = (int)(m2 / 2)`		80	**0**	65
23	`while (m2 != 0)`	This evaluates to `false`			
24	`.println(s)`	The value 65 is displayed which is, of course, the result of the multiplication 5 × 13			

Exercise 30.5-4 Finding the Number of Divisors

Write a Java program that lets the user enter a positive integer and then displays the total number of its divisors.

Solution

Let's see some examples.

- The divisors of value 12 are numbers 1, 2, 3, 4, 6, 12.
- The divisors of value 15 are numbers 1, 3, 5, 15.
- The divisors of value 20 are numbers 1, 2, 4, 5, 10, 20.
- The divisors of value 50 are numbers 1, 2, 5, 10, 25, 50.

If variable `x` contains the user-provided integer, all possible divisors of `x` are between 1 and `x`. Thus, all you need here is a for-loop where the value of variable *`counter`* goes from 1 to `x` and, in each iteration, a simple-alternative decision structure checks whether the value of *`counter`* is a divisor of `x`, The Java program is as follows.

```
Project_30.5-4a
public static void main(String[] args) throws Exception {
                int x, numberOfDivisors, i;
            x = Integer.parseInt(cin.nextLine());
        numberOfDivisors = 0; for (i = 1; i <= x; i++) {
```

```
                if (x % i == 0) {
                 numberOfDivisors++;
                        }
                      }
        System.out.println(numberOfDivisors); }
```

This program, for input value 20, performs 20 iterations. However, wouldn't it be even better if it could perform less than the half of the iterations and achieve the same result? Of course it would! So, let's make it more efficient!

As you probably know, for any user-provided integer (in variable x) ► the value 1 is always a divisor.

- ► the user-provided integer is always a divisor of itself.
- ► except for the user-provided integer, there are no other divisors after the middle of the range 1 to x.

Accordingly, for any integer there are certainly 2 divisors, the value 1 and the user-provided integer itself. Therefore, the program must check for other possible divisors starting from the value 2 until the middle of the range 1 to x. The improved Java program is as follows.

```
                🗀 Project_30.5-4b
    public static void main(String[] args) throws Exception {
                    int x, numberOfDivisors, i;
                x = Integer.parseInt(cin.nextLine());
     numberOfDivisors = 2; for (i = 2; i <= (int)(x / 2); i++) {
                    if (x % i == 0) {
                     numberOfDivisors++;
                            }
                          }
            System.out.println(numberOfDivisors); }
```

This Java program performs less than half of the iterations that the previous program did! For example, for the input value 20, this Java program performs only (20 − 2) DIV 2 = 9 iterations!

Exercise 30.5-5 Is the Number a Prime?

Write a Java program that prompts the user to enter an integer greater than 1 and then displays a message indicating if this number is a prime. A prime number is any integer greater than 1 that has no divisors other than 1 and itself. The numbers 7, 11, and 13 are all such numbers.

Solution

This exercise is based on the previous one. It is very simple! If the user-provided integer has only two divisors (1 and itself), the number is a prime. The Java program is as follows.

```
                🗀 Project_30.5-5a
    public static void main(String[] args) throws Exception {
                    int x, numberOfDivisors, i;
     System.out.print("Enter an integer greater than 1: "); x =
                Integer.parseInt(cin.nextLine());
     numberOfDivisors = 2; for (i = 2; i <= (int)(x / 2); i++) {
                    if (x % i == 0) {
                     numberOfDivisors++;
                            }
                          }
                if (numberOfDivisors == 2) {
            System.out.println("Number " + x + " is prime");
                            }
                         }
```

Now let's make the program more efficient. The flow of execution can break out of the loop when a third divisor is found, because this means that the user-provided integer is definitely not a prime. The Java program is as follows.

Project_30.5-5b

```
public static void main(String[] args) throws Exception {
                int x, numberOfDivisors, i;
 System.out.print("Enter an integer greater than 1: "); x =
              Integer.parseInt(cin.nextLine());
 numberOfDivisors = 2; for (i = 2; i <= (int)(x / 2); i++) {
                    if (x % i == 0) {
                     numberOfDivisors++;
                           break;
                      }
                    }
              if (numberOfDivisors == 2) {
          System.out.println("Number " + x + " is prime");
                    }
                  }
```

Exercise 30.5-6 Finding all Prime Numbers from 1 to N

Write a Java program that prompts the user to enter an integer greater than 1 and then displays all prime numbers from 1 to that user-provided integer. Using a loop control structure, the program must also validate data input and display an error message when the user enters any values less than 1.

Solution

The following Java program, given in general form, solves this exercise.

Main Code

```
System.out.print("Enter an integer greater than 1 "); N =
   Integer.parseInt(cin.nextLine()); while (N <= 1) {
System.out.print("Wrong number. Enter an integer greater than 1: "); N =
   Integer.parseInt(cin.nextLine()); }
   for (x = 1; x <= N; x++) {

   Code Fragment 1: Check whether variable x
        contains a prime number

                  }
```

Code Fragment 1, shown below, is taken from the previous exercise (Exercise 30.5-5). It checks whether variable x contains a prime number.

Code Fragment 1

```
numberOfDivisors = 2;
for (i = 2; i <= (int)(x / 2); i++) {
   if (x % i == 0) {
      numberOfDivisors++;
      break;
   }
}
if (numberOfDivisors == 2) {
   System.out.println("Number " + x + " is prime"); }
```

After embedding **Code Fragment 1** in **Main Code**, the final Java program becomes Project_30.5-6

```
public static void main(String[] args) throws Exception {
   int N, x, numberOfDivisors, i;
   System.out.print("Enter an integer greater than 1: "); N = Integer.parseInt(cin.nextLine()); while (N <= 1) {
      System.out.print("Wrong number. Enter an integer greater than 1: ");
      N = Integer.parseInt(cin.nextLine());
   }
   for (x = 1; x <= N; x++) {
```

```
      numberOfDivisors = 2; [More…]
      for (i = 2; i <= (int)(x / 2); i++) {
         if (x % i == 0) {
            numberOfDivisors++;
            break;
         }
      }
      if (numberOfDivisors == 2) {
         System.out.println("Number " + x + " is prime");
      }
   }
}
```

Exercise 30.5-7 Heron's Square Root

Write a Java program that prompts the user to enter a non-negative value and then calculates its square root using Heron's formula, as follows.

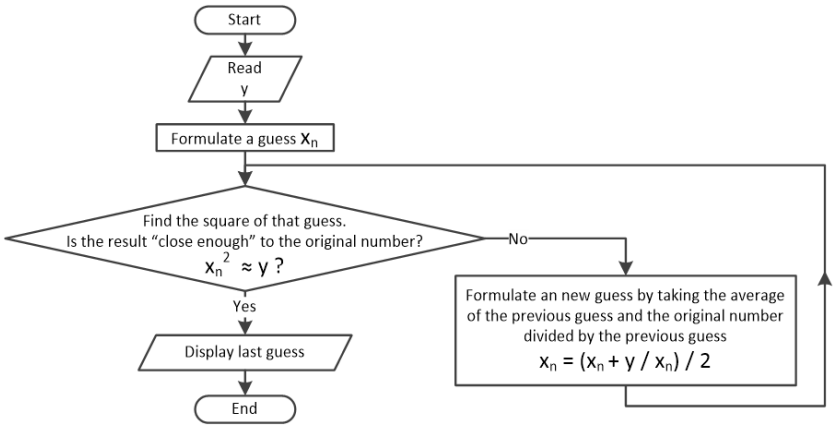
$$x_{n+1} = \frac{\left(x_n + \frac{y}{x_n}\right)}{2}$$

where

- ***y** is the number for which you want to find the square root* ***x_n** is the n-th iteration value of the square root of y* Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any negative values.

Solution

It is almost certain that you are a little bit confused and you are scratching your head right now. Don't get scared by all this math stuff! You can try to understand Heron's formula through the following flowchart instead!


Start
Read y
Formulate a guess x_n
Find the square of that guess.
Is the result "close enough" to the original number?
$x_n^2 \approx y$?
No
Formulate an new guess by taking the average of the previous guess and the original number divided by the previous guess
$x_n = (x_n + y / x_n) / 2$
Yes
Display last guess
End

Still confused? Let's go through an example. Let's try to find the square root of 25: Formulate a guess. Assume 8 as your first guess.

- The square of 8 is 64.
- Since 64 isn't "close enough" to 25, formulate a new guess by calculating the expression

$$\frac{\left(guess + \frac{number}{guess}\right)}{2} = \frac{\left(8 + \frac{25}{8}\right)}{2} \approx 5.56$$

  - The square of 5.56 is about 30.91
  - Since 30.91 isn't "close enough" to 25, formulate a new guess by calculating the expression

$$\frac{\left(guess + \frac{number}{guess}\right)}{2} = \frac{\left(5.56 + \frac{25}{5.56}\right)}{2} \approx 5.02$$

    - The square of 5.02 is 25.2

► If you think that 25.2 is “close enough” to 25, then you can stop the whole process and conclude that the approximate square root of 25 is 5.02.

✎ *Obviously, if greater precision is required, you have the option to continue the process until you find a value that is considered closer to the square root of 25.*

Now, let's see the corresponding Java program.

Project_30.5-7

```
static final double ACCURACY = 0.0000000000001;
public static void main(String[] args) throws Exception {
   double y, guess;
   System.out.print("Enter a non-negative number: "); y = Double.parseDouble(cin.nextLine()); while (y < 0) {
      System.out.print("Invalid value. Enter a non-negative number: ");
      y = Double.parseDouble(cin.nextLine());
   }
   //Make a random first guess between 1 and user-provided value guess = 1 + (Math.random() * y);
   while (Math.abs(guess * guess - y) > ACCURACY) { //Is it "close enough"?
      guess = (guess + y / guess) / 2; //No, create a new "guess"!
   }
   System.out.println(guess); }
```

✎ *Note the way that “Is it close enough” is checked. When the absolute value of the difference $|guess^2 - y|$ becomes less than 0.0000000000001 (where y is the user-provided value), the flow of execution exits the loop.*

Exercise 30.5-8 Calculating π

Write a Java program that calculates π using the Madhava–Leibniz[19],[20] series, which follows, with an accuracy of 0.00001.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Solution

The Madhava–Leibniz series can be solved for π, and becomes $\pi = \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots$

The more fractions you have, the better the accuracy! Thus, to calculate this formula the program needs to perform many iterations so as to use as many fractions as possible. But, of course, it can't iterate forever! The loop must actually stop iterating when the current calculated value of π and the one calculated in the previous iteration are ”close enough”, which means that the absolute value of their difference has become very small. The constant `ACCURACY` defines how small this difference must be. The Java program is shown here.

Project_30.5-8

```
static final double ACCURACY = 0.00001;
public static void main(String[] args) throws Exception {
   double pi, piPrevious; int sign, denom;
   pi = 0;
   sign = 1; //This is the sign of the first fraction denom = 1; //This is the denominator of the first fraction do
   {
      piPrevious = pi; //Keep previous pi
      pi += sign * 4 / (double)denom; //Calculate new pi by adding a fraction (a term)
      sign = -sign; //Prepare sign for the next fraction
      denom += 2; //Prepare denominator for the next fraction
   } while (Math.abs(pi - piPrevious) > ACCURACY); //Is it "close enough"?
   System.out.println("Pi ~= " + pi); }
```

✎ *Note the way in which variable* `sign` *toggles between the values −1 and +1 in each iteration.*

If you reduce the value of the constant `ACCURACY`, π will be calculated more and more accurately. Depending on how fast your computer is, you can calculate the first five digits of π fairly quickly.

However, the time it takes to calculate each succeeding digit of π goes up exponentially. To calculate 40 digits of π on a modern computer using this method could take years!

Exercise 30.5-9 Approximating a Real with a Fraction

Write a Java program that prompts the user to enter a real between 0 and 100 and then tries to find the fraction $\frac{N}{M}$ *that better approximates it, where N is an integer between 0 and 100 and M is an integer between 1 and 100. Using a loop control structure, the program must also validate data input, allowing the user to enter only values between 0 and 100. There is no need to display any error messages.*

Solution

The solution is simple. All you need to do is iterate through all possible combinations of variables `n` and `m` and check which one better approximates the user-provided real.

To iterate through all possible combinations of variables `n` and `m`, you can use a nested loop control structure, that is, two for-loops, one nested within the other, as follows.

```
for (n = 0; n <= 100; n++) {
   for (m = 1; m <= 100; m++) {
      …
   }
}
```

✎ *The total number of iterations is 101 × 100 = 10100. Quite a big number but, for a modern computer, this is peanuts!*

✎ *Variable* `m` *represents the denominator of the fraction, and a denominator cannot be zero. This is why it starts from 1, and not from 0.*

The following criteria $minimum\ of\left(\left|\frac{N}{M} - user\text{-}provided\ real\right|\right)$ can evaluate how "good" an approximation is.

Confused? Let's try to approximate the value 0.333 with a fraction, iterating through all possible combinations of N and M.

- For N = 1, M = 1 the criteria equals to $\left|\frac{1}{1} - 0.333\right|$ = 0.6670
- For N = 1, M = 2 the criteria equals to $\left|\frac{1}{2} - 0.333\right|$ = 0.1670
- For N = 1, M = 3 the criteria equals to $\left|\frac{1}{3} - 0.333\right|$ = 0.0003
- For N = 1, M = 4 the criteria equals to $\left|\frac{1}{4} - 0.333\right|$ = 0.0830
- …
- For N = 100, M = 99 the criteria equals to $\left|\frac{100}{99} - 0.333\right|$ = 0.6771
- For N = 100, M = 100 the criteria equals to $\left|\frac{100}{100} - 0.333\right|$ = 0.6670

It is obvious that the value 0.0003 is the minimum value among all possible results. Thus, the combination N = 1 and M = 3 (which corresponds to the fraction 1/3) is considered the best approximation for the value 0.333.

And now the Java program: Project_30.5-9

```
public static void main(String[] args) throws Exception {
   double x, y, minimum; int bestN, bestM, n, m;
   do {
      System.out.print("Enter a real between 0 and 100: ");
      x = Double.parseDouble(cin.nextLine());
   } while (x < 0 || x > 100);
```

```
bestN = 1;
bestM = 1;
minimum = 100;
for (n = 0; n <= 100; n++) {
   for (m = 1; m <= 100; m++) {
      y = Math.abs(n /(double)m - x);
      if (y < minimum) {
         minimum = y;
         bestN = n;
         bestM = m;
      }
   }
}
System.out.println("The fraction is: " + bestN + " / " + bestM); }
```

✎ *Converting variable* `m` *to type* `double` *tricks Java and a normal division is performed (a division that returns a real). If you omit the* `(double)` *casting operator, since both dividend and divisor are of type* `int`*, Java performs an integer division and produces incorrect results.*

30.6 Exercises of a General Nature with Loop Control Structures

Exercise 30.6-1 Fahrenheit to Kelvin, from 0 to 100

Write a Java program that displays all degrees Fahrenheit from 0 to 100 and their equivalent degrees Kelvin. Use an increment value of 0.5. It is given that 1.8 · Kelvin = Fahrenheit + 459.67

Solution

The formula, solved for Kelvin becomes $Kelvin = \frac{Fahrenheit + 459.67}{1.8}$

All you need here is a for-loop that increments the value of variable `fahrenheit` from 0 to 100 using an *offset* of 0.5. The solution is presented next.

Project_30.6-1

```
public static void main(String[] args) throws Exception {
   double fahrenheit, kelvin;
   for (fahrenheit = 0; fahrenheit <= 100; fahrenheit += 0.5) {
      kelvin = (fahrenheit + 459.67) / 1.8;
      System.out.println("Fahrenheit: " + fahrenheit + " Kelvin: " + kelvin);
   }
}
```

Exercise 30.6-2 Rice on a Chessboard

There is a myth about a poor man who invented chess. The King of India was so pleased with that new game that he offered to give the poor man anything he wished for. The poor but wise man told his King that he would like one grain of rice for the first square of the board, two grains for the second, four grains for the third and so on, doubled for each of the 64 squares of the game board. This seemed to the King to be a modest request, so he ordered his servants to bring the rice.

Write a Java program that calculates and displays how many grains of rice, and how many pounds of rice, will be on the chessboard in the end. Suppose that one pound of rice contains about 30,000 grains of rice.

Solution

Assume a chessboard of only 2 × 2 = 4 squares and a variable `grains` assigned the initial value 1 (this is the number of grains of the 1st square). A for-loop that iterates three times can double the value of variable `grains` in each iteration, as shown in the next code fragment.

```
grains = 1;
for (i = 2; i <= 4; i++) {
   grains = 2 * grains; }
```

The value of variable `grains` at the end of each iteration is shown in the next table.

Iteration	Value of `grains`

1st	2 × 1 = **2**
2nd	2 × 2 = **4**
3rd	2 × 4 = **8**

At the end of the 3rd iteration, variable `grains` contains the value 8. This value is not the total number of grains on the chessboard but only the number of grains on the 4th square. If you need to find the total number of grains on the chessboard you can sum up the grains on all squares, that is, 1 + 2 + 4 + 8 = 15.

In the real world a real chessboard contains 8 × 8 = 64 squares, thus you need to iterate for 63 times. The Java program is as follows.

Project_30.6-2

```
public static void main(String[] args) throws Exception {
   int i; double grains, total, weight;
   grains = 1;
   total = 1;
   for (i = 2; i <= 64; i++) {
      grains = 2 * grains;
      total = total + grains;
   }
   weight = total / 30000;
   System.out.println(total + " " + weight); }
```

In case you are wondering how big these numbers are, here is your answer: On the chessboard there will be 18,446,744,073,709,551,615 grains of rice; that is, 614,891,469,123,651.8 pounds!

Exercise 30.6-3 Just a Poll

A public opinion polling company asks 1000 citizens if they eat breakfast in the morning. Write a Java program that prompts the citizens to enter their gender (M for Male, F for Female, O for Other) and their answer to the question (Y for Yes, N for No, S for Sometimes), and then calculates and displays the number of citizens that gave "Yes" as an answer, as well as the percentage of women among the citizens that gave "No" as an answer. Using a loop control structure, the program must also validate data input and accept only values M, F or O for gender and Y, N, or S for answer.

Solution

The Java program is as follows.

Project_30.6-3

```
static final int CITIZENS = 1000;
public static void main(String[] args) throws Exception {
   int totalYes, femaleNo, i; String gender, answer;
   totalYes = 0;
   femaleNo = 0;
   for (i = 1; i <= CITIZENS; i++) {
      do {
         System.out.print("Enter gender: ");
         gender = cin.nextLine().toLowerCase();
      } while (!gender.equals("m") && !gender.equals("f") && !gender.equals("o"));
      do {
         System.out.print("Do you eat breakfast in the morning? ");
         answer = cin.nextLine().toLowerCase();
      } while (!answer.equals("y") && !answer.equals("n") && !answer.equals("s"));
      if (answer.equals("y")) {
         totalYes++;
      }
      if (gender.equals("f") && answer.equals("n")) {
         femaleNo++;
      }
   }
   System.out.println(totalYes + " " + femaleNo * 100 /(double)CITIZENS + "%"); }
```

✎ *Note how Java converts the user's input to lowercase.*

Exercise 30.6-4 Is the Message a Palindrome?

A palindrome is a word or sentence that reads the same both backwards and forward. (You may recall from Exercise 23.5-4 that a number can also be a palindrome). Write a Java program that prompts the user to enter a word or sentence and then displays a message stating whether or not the user-provided word or sentence is a palindrome. Following are some palindrome words and messages.

- *Anna Radar Madam A nut for a jar of tuna.*
- *Dennis and Edna sinned.*
- *Murder for a jar of red rum.*
- *Borrow or rob?*
- *Are we not drawn onward, we few, drawn onward to new era?*

Solution

There are some things you should keep in mind before starting to compare the letters one by one and checking whether the first letter is the same as the last one, the second letter is the same as the last but one, and so forth.

- In a given sentence or word, some letters may be in uppercase and some in lowercase. For example, in the sentence "*A nut for a jar of tuna*", even though the first and last letters are the same, they are not considered equal. Thus, the program must first convert all the letters—for example, to lowercase—before it can start comparing them.
- Removing characters like spaces, periods, question marks, and commas is crucial for the program to accurately compare the letters. For example, without this step, in the sentence "*Borrow or rob?*" the program will mistakenly assume it's not a palindrome, as it would attempt to compare the initial "B" with the final question mark "?".
- Assume that the examined sentence is "*Borrow or rob?*". After changing all letters to lowercase and after removing all unwanted spaces and the question mark, the sentence becomes "*borroworrob*". These letters and their corresponding position in the string are as follows:

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
b	**o**	**r**	**r**	**o**	**w**	**o**	**r**	**r**	**o**	**b**

  What you should realize here is that the for-loop should iterate for only half of the letters. Can you figure out why?

  The program should start the iterations and compare the letter at position 0 with the letter at position 10. Then it should compare the letter at position 1 with the letter at position 9, and so forth. The last iteration should be the one that compares the letters at positions 4 and 6. It would be pointless to continue checking thereafter, since all letters have already been compared.

There are many solutions to this problem. Some of them are presented below. Comments written within the programs can help you fully understand the way they operate. However, if you still have doubts about how they operate you can use Visual Studio Code to execute them step by step and observe the values of the variables in each step.

First approach

The solution is presented here.

```
Project_30.6-4a
    public static void main(String[] args) throws Exception {
int i, middlePos, j; String message, messageClean, letter; char
        leftLetter, rightLetter; boolean palindrome;
        System.out.print("Enter a message: "); message =
                cin.nextLine().toLowerCase();
//Create a new string which contains all except spaces, commas, periods
            and question marks messageClean = "";
          for (i = 0; i <= message.length() - 1; i++) {
                letter = "" + message.charAt(i);
```

```
            if (!letter.equals(" ") && !letter.equals(",") &&
                !letter.equals(".") && !letter.equals("?")) {
                    messageClean += letter;
                    }
                }
    j = messageClean.length() - 1; //This is the last position of
messageClean middlePos = (int)(j / 2); //This is the middle position of
                    messageClean
    palindrome = true; //In the beginning, assume that sentence is
                    palindrome
            //This for-loop compares letters one by one.
                for (i = 0; i <= middlePos; i++) {
                leftLetter = messageClean.charAt(i);
                rightLetter = messageClean.charAt(j);
    //If at least one pair of letters fails to validate set variable
                    palindrome to false
                if (leftLetter != rightLetter) {
                    palindrome = false;
                    }
                j--;
                }
      //If variable palindrome is still true if (palindrome) {
            System.out.println("The message is palindrome");
                    }
                }
```

Second approach

The previous approach works fine, but let's assume that the user enters a very large sentence that is not a palindrome; for example, its second letter is not the same as the last but one. Unfortunately, in the previous approach, the last for-loop continues to iterate until the middle of the sentence despite the fact that the variable `palindrome` has been set to `false`, even from the second iteration. So, let's try to make this program even better. As you already know, you can break out of a loop before it completes all of its iterations using the `break` statement.

Furthermore, since there are just four different characters that must be removed (spaces, commas, periods, and question marks) you can avoid the first loop if you just chain four `replace()` methods, as shown in the Java program that follows.

```
                    Project_30.6-4b
    public static void main(String[] args) throws Exception {
int i, middlePos, j; String message, messageClean; boolean palindrome;
            System.out.print("Enter a message: "); message =
                    cin.nextLine().toLowerCase();
//Create a new string which contains all except spaces, commas, periods
and question marks messageClean = message.replace(" ", "").replace(",",
                "").replace(".", "").replace("?", "");
        j = messageClean.length() - 1; middlePos = (int)(j / 2);
                    palindrome = true;
                for (i = 0; i <= middlePos; i++) {
          if (messageClean.charAt(i) != messageClean.charAt(j)) {
                    palindrome = false;
                        break;
                    }
                j--;
                }
                if (palindrome) {
            System.out.println("The message is palindrome");
```

```
        }
    }
```

✎ *It is obvious that one problem can have many solutions. It is up to you to find the optimal one!*

✎ *If you wish to remove **all** the unwanted characters (spaces, commas, periods, question marks, ampersands, etc.), you can use the following code fragment instead. It keeps only the letters in the variable* `messageClean`*!*

```
//Create a new string which contains only letters messageClean = "";
String validChars = "abcdefghijklmnopqrstuvwxyz"; for (i = 0; i < message.length(); i++) {
   if (validChars.indexOf(message.charAt(i)) != -1) {
      messageClean += message.charAt(i); //Concatenation
   }
}
```

30.7 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Data validation is the process of restricting data input, forcing the user to enter only valid values.
2) You can use a definite loop to validate data input.
3) To force a user to enter only positive numbers, without displaying any error messages, you can use the following code fragment.

```
do {
   System.out.print("Enter a positive number: "); x = Double.parseDouble(cin.nextLine()); } while (x <= 0);
```

4) To force a user to enter numbers between 1 and 10, you can use the following code fragment.

```
System.out.print("Enter a number between 1 and 10: "); x = Double.parseDouble(cin.nextLine()); while (x >= 1
&& x <= 10) {
   System.out.println("Wrong number"); System.out.print("Enter a number between 1 and 10: "); x =
   Double.parseDouble(cin.nextLine()); }
```

5) In order to find the lowest number among 10 user-provided numbers, you can use the following code fragment.

```
minimum = 0;
for (i = 1; i <= 10; i++) {
   w = Double.parseDouble(cin.nextLine()); if (w < minimum)
   minimum = w;
}
```

6) In order to find the highest number among 10 user-provided numbers, you can use the following code fragment.

```
maximum = 0;
for (i = 1; i <= 10; i++) {
   w = Double.parseDouble(cin.nextLine()); if (w > maximum) {
      maximum = w;
   }
}
```

7) In order to find the highest number among 10 positive user-provided numbers, you can use the following code fragment.

```
maximum = 0;
for (i = 1; i <= 10; i++) {
   w = Double.parseDouble(cin.nextLine()); if (w > maximum) {
      maximum = w;
   }
}
```

30.8 Review Exercises

Complete the following exercises.

1) Design a flowchart and write the corresponding Java program that prompts the user to repeatedly enter non-negative values until their average value exceeds 3000. At the end, the program must display the total number of zeros entered.

2) Write a Java program that prompts the user to enter an integer between 1 and 20 and then displays all four-digit integers for which the sum of their digits is less than the user-provided integer. For example, if the user enters 15, the value 9301 is such a number, since 9 + 3 + 0 + 1 < 15

3) Write a Java program that displays all four-digit integers that satisfy all of the following conditions: the number's first digit is greater than its second digit the number's second digit is equal to its third digit the number's third digit is smaller than its fourth digit For example, the values 7559, 3112, and 9889 are such numbers.

4) Write a Java program that prompts the user to enter an integer and then displays the number of its digits.

5) A student wrote the following code fragment which is supposed to validate data input, forcing the user to enter only values 0 and 1. Identify any error(s) in the code fragment.

```
while (x != 1 || x != 0) {
   System.out.println("Error"); x = Integer.parseInt(cin.nextLine()); }
```

6) Using a loop control structure, write the code fragment that validates data input, forcing the user to enter a valid gender (M for Male, F for Female, O for Other). Moreover, it must validate correctly both for lowercase and uppercase letters.

7) Write a Java program that prompts the user to enter a non-negative number and then calculates its square root. Using a loop control structure, the program must also validate data input and display an error message when the user enters any negative values. Additionally, the user has a maximum number of two retries. If the user enters more than three negative values, a message “Dude, you are dumb!” must be displayed and the program execution must end.

8) The area of a circle can be calculated using the following formula: $Area = \pi \cdot Radius^2$

Write a Java program that prompts the user to enter the length of the radius of a circle and then calculates and displays its area. The program must iterate as many times as the user wishes. At the end of each area calculation, the program must ask the user if they wish to calculate the area of another circle. If the answer is “yes” the program must repeat; it must end otherwise. Make your program accept the answer in all possible forms such as “yes”, “YES”, “Yes”, or even “YeS”.

Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any non-positive value for *Radius*.

Hint: Use the `Math.PI` constant to get the value of π.

9) Write a Java program that prompts the user to enter the daily temperatures (in degrees Fahrenheit) recorded at the same hour each day in August and then calculates and displays the average as well as the highest temperature.

Since -459.67^{o} (on the Fahrenheit scale) is the lowest temperature possible (it is called absolute zero), using a loop control structure, the program must also validate data input and display an error message when the user enters a value lower than absolute zero.

10) A scientist needs a software application to record the level of the sea based on values logged at specific times (HH:MM), in order to extract some useful information. Write a Java program that lets the scientist enter the sea level, along with the hour and minutes, repeatedly until the value 9999 is entered for the sea level. Then, the program must display both the highest and the lowest recorded sea levels, along with the corresponding hour and minutes at which these levels were recorded.

11) In some countries, when someone sneezes, a number (an integer) is said aloud by another person. The sneezing person then adds up the digits of this number until they obtain a number between 1 and 26. The letter

corresponding to this number (1 for “A”, 2 for “B”, and so on) represents the first letter of the name of someone who might be thinking of them.

Write a Java program that prompts the user to enter the number said after the sneeze. It must then sum up the digits of the number until a number between 1 and 26 is obtained, and display the corresponding letter in the English alphabet.

12) Write a Java program that displays all possible integer values of *x* and *y* within the range −100 to +100 that validate the following formula: $5x + 3y^2 = 0$

13) Write a Java program that displays all possible integer values of *x*, *y*, and *z* within the range −10 to +10 that validate the following formula:

$$\frac{x+y}{2} + \frac{3z^2}{x+3y+45} = \frac{x}{3}$$

14) Write a Java program that lets the user enter three positive integers and then finds their product using the Russian multiplication algorithm.

15) Rewrite the Java program of Exercise 30.5-4 to validate the data input using a loop control structure. If the user enters a non-positive integer, an error message must be displayed.

16) Rewrite the Java program of Exercise 30.5-5 to validate the data input using a loop control structure. If the user enters an integer less than or equal to 1, an error message must be displayed.

17) Write a Java program that prompts the user to enter two positive integers into variables `start` and `finish`. The program must then find and display all Pythagorean triples (x, y, z) where x, y, and z are integers between `start` and `finish` such that $x^2 + y^2 = z^2$.

    Hint: To make your program operate correctly, independent of which user-provided integer is the lowest, you can swap their values (if necessary) so that they are always in the proper order.

18) Write a Java program that prompts the user to enter two positive integers and then displays all prime integers between them. Using a loop control structure, the program must also validate data input and display an error message when the user enters a value less than +2.

    Hint: To make your program operate correctly, independent of which user-provided integer is the lowest, you can swap their values (if necessary) so that they are always in the proper order.

19) A perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For example, the divisors of 6 are 1, 2, and 3 (excluding 6 itself), and 1 + 2 + 3 = 6, making 6 a perfect number. Write a Java program that prompts the user to enter a positive integer and displays a message indicating whether or not the number is perfect. Using a loop control structure, the program must also validate data input and display an error message when the user enters a non-positive integer.

20) Write a Java program that prompts the user to enter two positive integers and then displays all perfect numbers between them. Using a loop control structure, the program must also validate data input and display an error message when the user enters a non-positive integer.

Hint: To make your program operate correctly, independent of which user-provided integer is the lowest, you can swap their values (if necessary) so that they are always in the proper order.

21) Write a Java program that prompts the user to enter two positive four-digit integers and then displays all integers between them that are palindromes. Using a loop control structure, the program must also validate data input and display an error message when the user enters any numbers other than four-digit ones.

Hint: To make your Java program operate correctly, independent of which user-provided integer is the lowest, you can swap their values (if necessary) so that they are always in the proper order.

22) Write a Java program that displays all possible RAM sizes between 1 byte and 1GByte, such as 1, 2, 4, 8, 16, 32, 64, 128, and so on.

Hint: 1GByte equals 2^{30} bytes, or 1073741824 bytes.

23) Write a Java program that displays the following sequence of numbers: 1, 11, 23, 37, 53, 71, 91, 113, 137, … 401

24) Write a Java program that displays the following sequence of numbers: −1, 1, −2, 2, −3, 3, −4, 4, … −100, 100

25) Write a Java program that displays the following sequence of numbers: 1, 11, 111, 1111, 11111, … 11111111

26) The Fibonacci[21] sequence is a series of numbers in the following sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

By definition, the first two terms are 0 and 1 and each subsequent term is the sum of the previous two.

Write a Java program that lets the user enter a positive integer and then displays as many Fibonacci terms as that user-provided integer.

27) Write a Java program that lets the user enter a positive integer and then displays all Fibonacci terms that are less than that user-provided integer.

28) Write a Java program that prompts the user to enter a positive integer N and then finds and displays the value of *y* in the following formula: $y = \frac{2 + 4 + 6 + \ ... \ + 2N}{1 \cdot 2 \cdot 3 \cdot 4 \cdot ... \cdot N}$

Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters a value less than 1.

29) Write a Java program that prompts the user to enter a positive integer N and then finds and displays the value of *y* in the following formula

$$y = \frac{1 - 3 + 5 - 7 + \ ... \ + (2N \cdot$$
$$N$$

Moreover, using a loop control structure, the program must validate data input and display an

error message when the user enters a non-positive value.

30) Write a Java program that prompts the user to enter an integer N greater than 2 and then finds and displays the value of *y* in the following formula:

$$y = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{5} + \frac{1}{7} - \ldots$$

Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters a value that is less than or equal to 2.

Hint: Note that beyond the term 1 / 3 the subsequent denominators increment by 2.

31) Write a Java program that prompts the user to enter a positive integer N and then finds and displays the value of *y* in the following formula:

$$y = \frac{1}{1^N} + \frac{1}{2^{N-1}} + \frac{1}{3^{N-2}}$$

Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters a non-positive value.

32) In mathematics, the factorial of a non-negative integer N is the product of all positive integers less than or equal to N, and it is denoted by N! The factorial of 0 is, by definition, equal to 1. In mathematics, you can write

$$N! = \begin{cases} 1 \cdot 2 \cdot 3 \cdot \ldots \\ 1, \end{cases}$$

For example, the factorial of

5 is written as 5! and is equal to 1 × 2 × 3 × 4 × 5 = 120.

Write a Java program that prompts the user to enter a non-negative integer N and then calculates its factorial.

33) Write a Java program that lets the user enter a value for *x* and then calculates and displays the exponential function e^x using the Taylor[22] series, shown next, with an accuracy of 0.00001.

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} +$$

Hint: Keep in mind that $\frac{x^0}{0!} = 1$.

34) Write a Java program that lets the user enter a value for *x* and then calculates and displays the sine of *x* using the Taylor series, shown next, with an accuracy of 0.00001.

$$sinx = x - \frac{x^3}{3!} + \frac{x}{5}$$

Hint: Keep in mind that *x* is

in radians and $\frac{x^1}{1!} = x$.

35) Write a Java program that lets the user enter a value for *x* and then calculates and displays the cosine of *x* using the Taylor series, shown next, with an accuracy of 0.00001.

$$cosx = 1 - \frac{x^2}{2!} + \frac{x}{4}$$

Hint: Keep in mind that *x* is in radians and $\frac{x^0}{0!} = 1$.

36) Suppose that the letter A corresponds to the number 1, the letter B corresponds to the number 2, and so on. Write a Java program that prompts the user to enter two integers and then displays all alphabet letters that exist between them. For example, if the user enters 3 and 6, the program must display C, D, E, F. Using a loop control structure, the program must also validate data input and

display a different error message for each type of input error when the user enters any negative, or any value greater than 26.

Hint: To make your Java program operate correctly, independent of which user-provided integer is the lowest, you can swap their values (if necessary) so that they are always in the proper order.

37) Write a Java program that randomly selects an integer between 1 and 100 and assigns it to a variable. The program must then prompt the user to guess the number. If the user's guess is smaller than the secret number, the message "*Your guess is smaller than my secret number. Try again!*" must be displayed. If the user's guess is greater than

the secret number, the message "*Your guess is bigger than my secret number. Try again!*" must be displayed. This process must repeat until the user correctly guesses the secret number. Once the user guesses correctly, the message "*You found it!*" must be displayed, along with the total number of attempts made by the user.

38) Expand the previous exercise/game by making it operate for two players. The player that wins is the one that finds the random secret number in fewer attempts.

39) The size of a TV screen always refers to its diagonal measurement. For example, a 40-inch TV screen is 40 inches diagonally, from one corner on top to the other corner on bottom. The old TV screens had a width-to-

height aspect ratio of 4:3, which means that for every 3 inches in TV screen height, there were 4 inches in TV screen width. Today, most TV screens have a width-to-height aspect ratio of 16:9, which means that for every 9 inches in TV screen height there are 16 inches in TV screen width. Using these aspect ratios and the Pythagorean Theorem, you can easily determine that:

- **for all 4:3 TV screens** Width = Diagonal × 0.8

  Height = Diagonal × 0.6
- **for all 16:9 TV screens** Width = Diagonal × 0.87

  Height = Diagonal × 0.49

Write a Java program that displays the following menu: 1) 4/3 TV Screen 2) 16/9 TV Screen 3) Exit and prompts the user to enter a choice (of 1, 2, or 3)

as well as the diagonal screen size in inches. Then, the Java program must display the width and the height of the TV screen. This process must continue repeatedly, until the user selects choice 3 (Exit) from the menu.

40) Write a Java program that prompts a teacher to enter the total number of students, their grades, and their gender (M for Male, F for Female, O for Other), and then calculates and displays all of the following: a) the average value of those who got an “A” (90 - 100) b) the average value of those who got a “B” (80 - 89) c) the average value of boys who got an “A” (90 - 100) d) the total number of girls that got less than “B” e) the highest and lowest grade f) the average grade of the whole class Add

all necessary checks to make the program satisfy the property of definitenes s. Moreover, using a loop control structure, the program must validate data input and display an error message when the teacher enters any of the following:

- non-positive values for total number of students
- negatives, or values greater than 100 for student grades
- values other than M, F, or O for gender

41) Write a Java program that calculates and displays the discount that a customer

receives based on the amount of their order, according to the following table.

Amount	Discount
$0 < amount < $20	0%
$20 ≤ amount < $50	3%
$50 ≤ amount < $100	5%
$100 ≤ amount	10%

At the end of each discount calculation, the program must ask the user if they wish to calculate the discount of another amount. If the answer is “yes”, the program must repeat; it must end otherwise. Make your program accept the answer in all possible forms such as “yes”, “YES”, “Yes”, or even “YeS”.

Moreover, using a loop control structure the program must validate data input and display an

error message when the user enters any non-positive value for amount.

42) The LAV Electricity Company charges subscribers for their electricity consumption according to the following table (monthly rates for domestic accounts).

Kilowatt-hours (kWh)	US pe kW
0 ≤ kWh ≤ 400	$0.1
401 ≤ kWh ≤ 1500	$0.2
1501 ≤ kWh ≤ 3500	$0.2
3501 ≤ kWh	$0.5

Write a Java program that prompts the user to enter the total number of kWh consumed by a subscriber and then calculates and displays the total amount to pay. This process must repeat until the value −1 for kWh is entered.

Moreover, using a loop control structure, the program must validate data

input and display an error message when the user enters any negative value for kWh. An exception for the value −1 must be made.

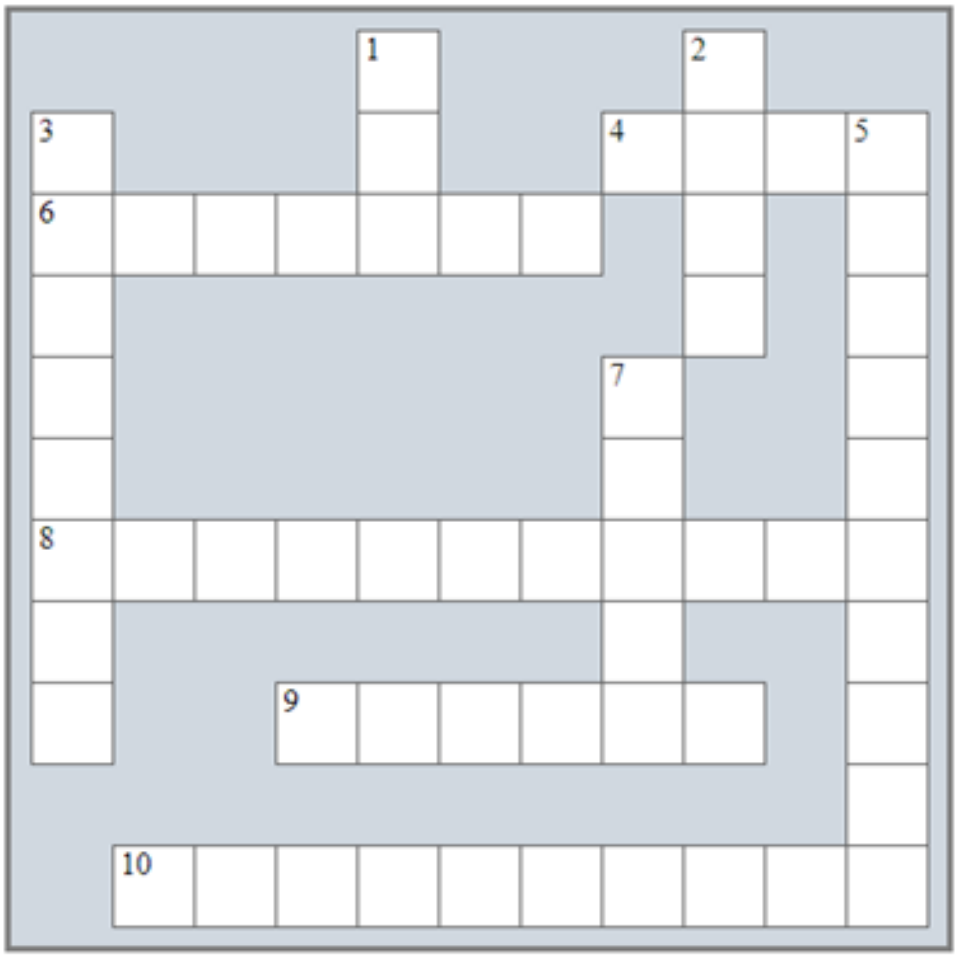
Transmission services and distribution charges, as well as federal, state, and local taxes, add a total of 25% to each bill.

Please note that the rates are progressive.

Review in “Loop Control Structures”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

4) This control structure allows the execution of a block of statements multiple times.

6) A loop that cannot stop iterating.

8) The "Ultimate" rule states that the variable that participates in a loop's Boolean expression must be _____ before entering the loop.

9) A loop within another loop.

10) In this loop structure, the number of iterations is not known before the loop starts iterating.

Down

1) In a _____–test loop structure, first the Boolean expression is evaluated, and afterward the statement or block of statements of the structure is executed.

2) The _____–test loop performs at least one iteration.

3) In this loop structure, the number of iterations is known before the loop starts iterating.

5) A word or sentence that reads the same both backward and forward.

7) Any integer greater than 1 that has no divisors other than 1 and itself.

Review Questions

Answer the following questions.

1) What is a loop control structure?
2) In a flowchart, how can you distinguish a decision control structure from a loop control structure?
3) Design the flowchart and write the Java statement (in general form) of a pre-test loop structure. Explain how this loop control structure operates.
4) Why is a pre-test loop structure named this way, and what is the fewest number of iterations it may perform?
5) If the statement or block of statements of a pre-test loop structure is executed N times, how many times is the Boolean expression of the structure evaluated?
6) Design the flowchart and write the corresponding Java statement (in general form) of a post-test loop structure. Explain how this loop control structure operates.
7) Why is a post-test loop structure named this way, and what is the fewest number of iterations it may perform?
8) If the statement or block of statements of a post-test loop structure is executed N times, how many times is the Boolean expression of the structure evaluated?

9) Design the flowchart and write the corresponding Java statement (in general form) of a mid-test loop structure. Explain how this loop control structure operates.
10) Design the flowchart and write the corresponding Java statement (in general form) of a for-loop. Explain how this loop control structure operates.
11) State the rules that apply to for-loops.
12) What are nested loops?
13) Write an example program that uses nested loop control structures and explain the way they are executed.
14) State the rules that apply to nested loops.
15) Design a diagram that could help someone decide which loop control structure is most appropriate to choose, depending on a given problem.
16) Describe the “Ultimate” rule and give two examples, in general form, using a pre-test and a post-test loop structure.
17) Suppose a Java program uses a loop control structure to search for a given word in an electronic English dictionary. Why is it critical to break out of the loop when the given word is found?
18) Why is it critical to clean out your loops?
19) What is an infinite loop?

Part VI
Data Structures in Java

Chapter 31
One-Dimensional Arrays and HashMaps

31.1 Introduction

Variables are a good way to store values in memory but they have one limitation—they can hold only one value at a time. There are many cases, however, where a program needs to keep a large amount of data in memory, and variables are not the best choice.

For example, consider the following exercise:

Write a Java program that lets the user enter three numbers. It then displays them sorted in ascending order.

Consider the following code fragment. It lets the user enter the three numbers.

```
for (i = 0; i <= 2; i++) {
   number = Double.parseDouble(cin.nextLine());
}
```

When the loop finally finishes iterating, the variable `number` contains only that last number that was provided. Unfortunately, all the previous two numbers have been lost! Using this code fragment, it is not quite possible to display them sorted in ascending order.

One possible solution would be to use three individual variables, as follows.

```
num1 = Double.parseDouble(cin.nextLine());
num2 = Double.parseDouble(cin.nextLine());
num3 = Double.parseDouble(cin.nextLine());
if (num1 <= num2 && num2 <= num3)
   System.out.println(num1 + " " + num2 + " " + num3);
else if (num1 <= num3 && num3 <= num2)
   System.out.println(num1 + " " + num3 + " " + num2);
else if (num2 <= num1 && num1 <= num3)
   System.out.println(num2 + " " + num1 + " " + num3);
else if (num2 <= num3 && num3 <= num1)
   System.out.println(num2 + " " + num3 + " " + num1);
else if (num3 <= num1 && num1 <= num2)
   System.out.println(num3 + " " + num1 + " " + num2);
else
   System.out.println(num3 + " " + num2 + " " + num1);
```

Not a perfect solution, but it works! However, what if the wording of this exercise asked the user to enter 1,000 numbers instead of three? Think about

it! Can you write a similar Java program for all those numbers? Of course not! Fortunately, there are *data structures*!

> ✎ *In computer science, a data structure is a collection of data organized so that you can perform operations on it in the most effective way.*

There are several data structures available in Java, such as *arrays*, *linked lists*, *stacks*, *queues*, *binary trees*, *heaps*, *hashtables*, *hashmaps*, and *strings*. Yes, you heard that right! Since a string is a collection of alphanumeric characters, it is considered a data structure.

Beyond strings (for which you have already learned enough), arrays and hashmaps are the most commonly used data structures in Java. The following chapters will analyze both of them.

31.2 What is an Array?

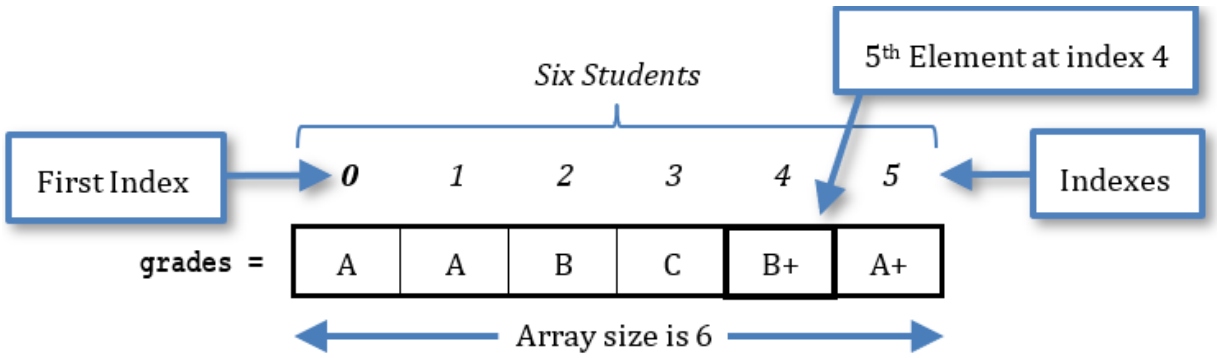
An *array* is a type of data structure that can hold multiple values under one common name. It can be thought of as a collection of *elements* where each element is assigned a unique number known as an *index position*, or simply an *index*. Arrays are *mutable* (changeable), which means that once an array is created, the values of its elements can be changed.

> ✎ *Arrays in computer science resemble the matrices used in mathematics. A mathematical matrix is a collection of numbers or other mathematical objects, arranged in rows and columns.*

There are one-dimensional and multidimensional arrays. A multidimensional array can be two-dimensional, three-dimensional, four-dimensional, and so on.

One-Dimensional Arrays

The following example presents a one-dimensional array that holds the grades of six students. The name of the array is `grades`. For your convenience, the corresponding index is written above each element. By default, in Java, index numbering always starts at zero.

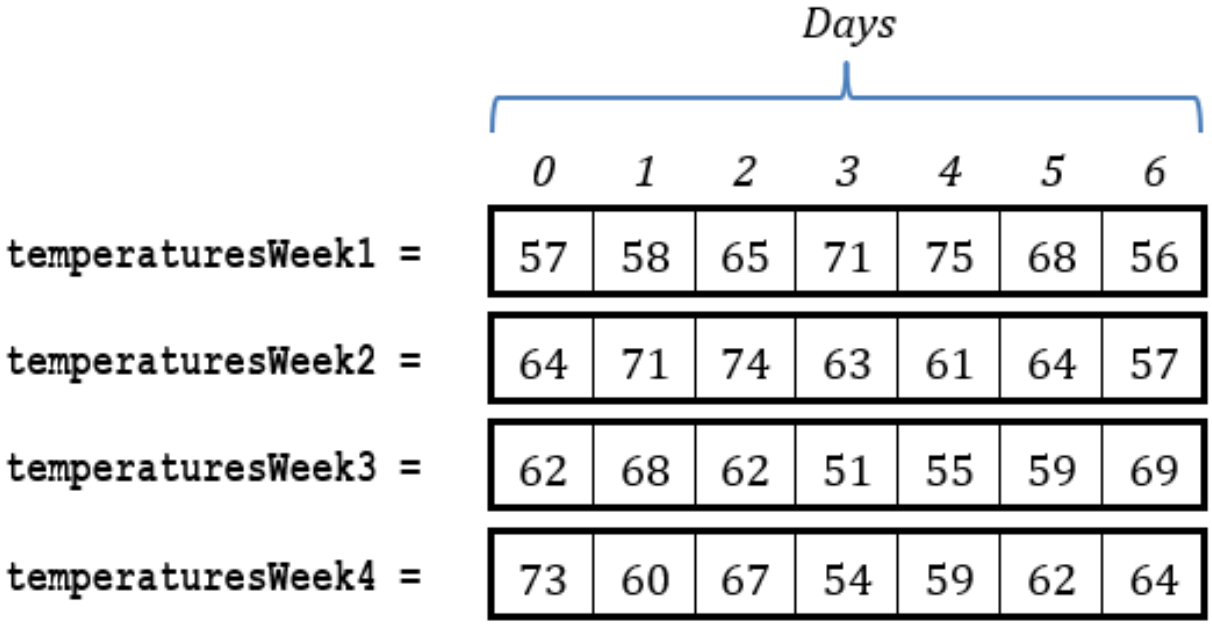


Since index numbering starts at zero, the index of the last element of an array is 1 less than the total number of elements in the array. In the array `grades`*, the index of the last element is 5 while the total number of elements is 6.*

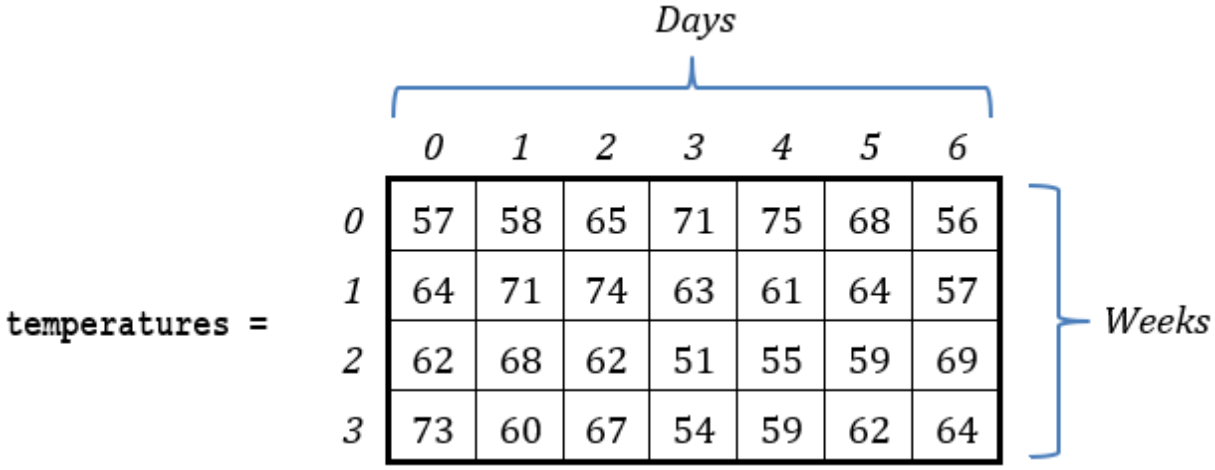
You can think of an array as if it were six individual variables—`grades0`, `grades1`, `grades2`, ... `grades5`—with each variable holding the grade of one student. The advantage of the array, however, is that it can hold multiple values under one common name.

Two-Dimensional Arrays

In general, multidimensional arrays are useful for working with multiple sets of data. For example, suppose you want to hold the daily high temperatures for California for the four weeks of April. One approach would be to use four one-dimensional arrays, one for each week. Furthermore, each array would have seven elements, one for each day of the week, as follows.

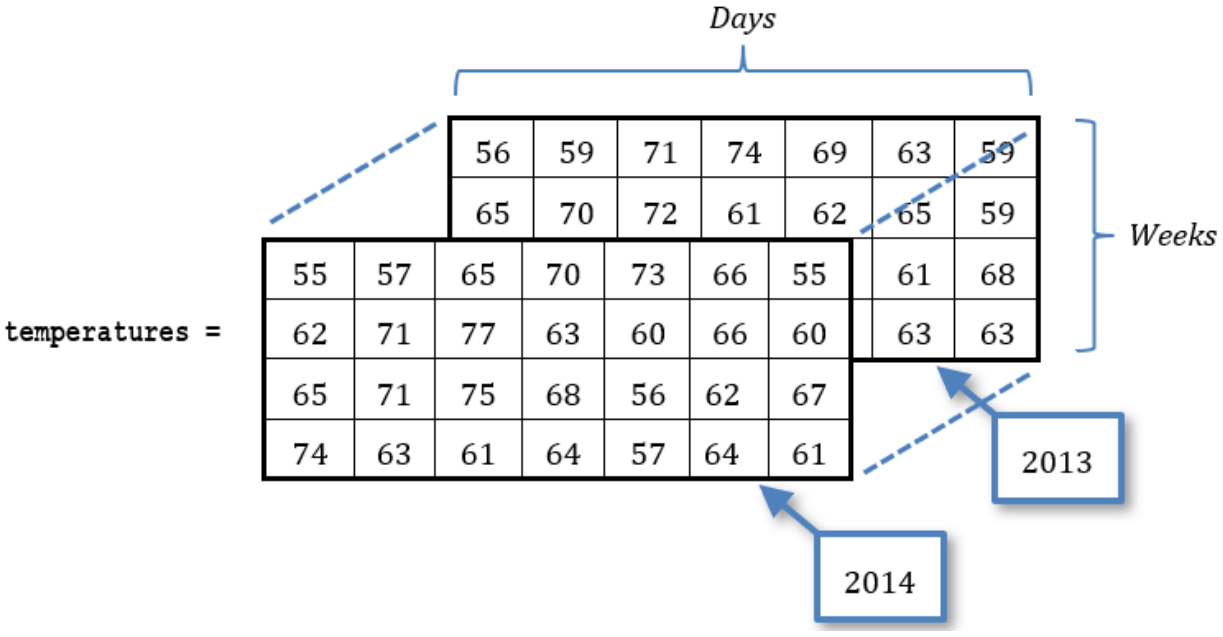


However, this approach is a bit awkward because you would have to process each array separately. A better approach would be to use a two-dimensional array with four rows (one for each week) and seven columns (one for each day of the week), as follows.



Three-Dimensional Arrays

The next example shows a three-dimensional array that holds the daily high temperatures for California for the four weeks of April for the years 2013 and 2014.



✎ *Note that four-dimensional, five-dimensional, or even one-hundred–dimensional arrays can exist. However, experience shows that the maximum array dimension that you will need in your life as a programmer is probably two or three.*

Exercise 31.2-1 Designing an Array

Design an array that can hold the ages of 8 people, and then add some typical values to the array.

Solution

This is an easy one. All you have to do is design an array with 8 elements (indexes 0 to 7). It can be an array with either one row or one column, as follows.

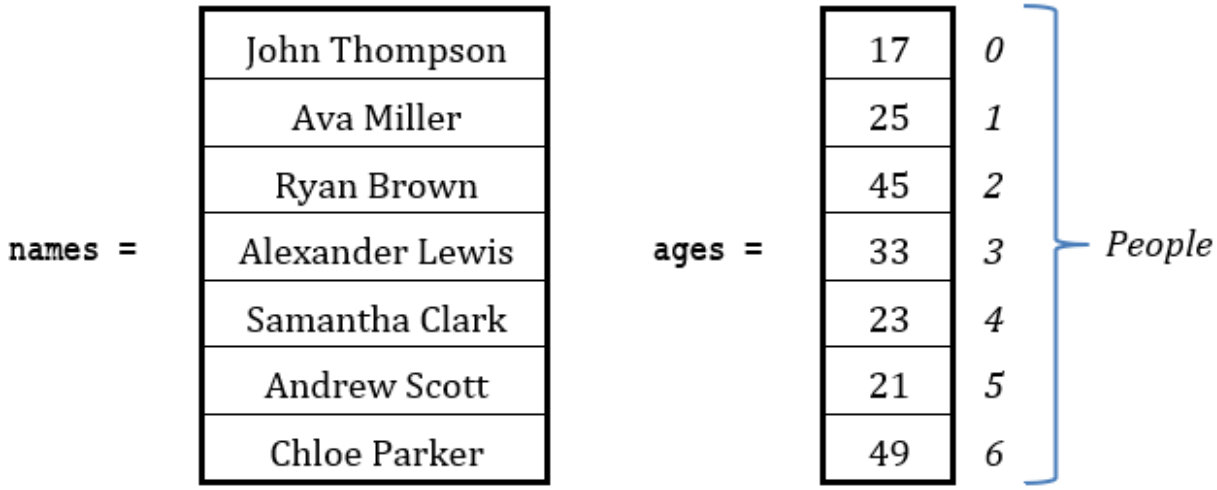
Keep in mind, however, that there are no arrays with one row or one column in Java. These concepts may exist in mathematical matrices (or in your imagination!) but not in Java. The arrays in Java are one-dimensional—end of story! If you want to visualize them having one row or one column, that is up to you.

Exercise 31.2-2 Designing Arrays

Design the necessary arrays to hold the names and the ages of seven people, and then add some typical values to the arrays.

Solution

This exercise can be implemented with two arrays. Let's design them with one column each.



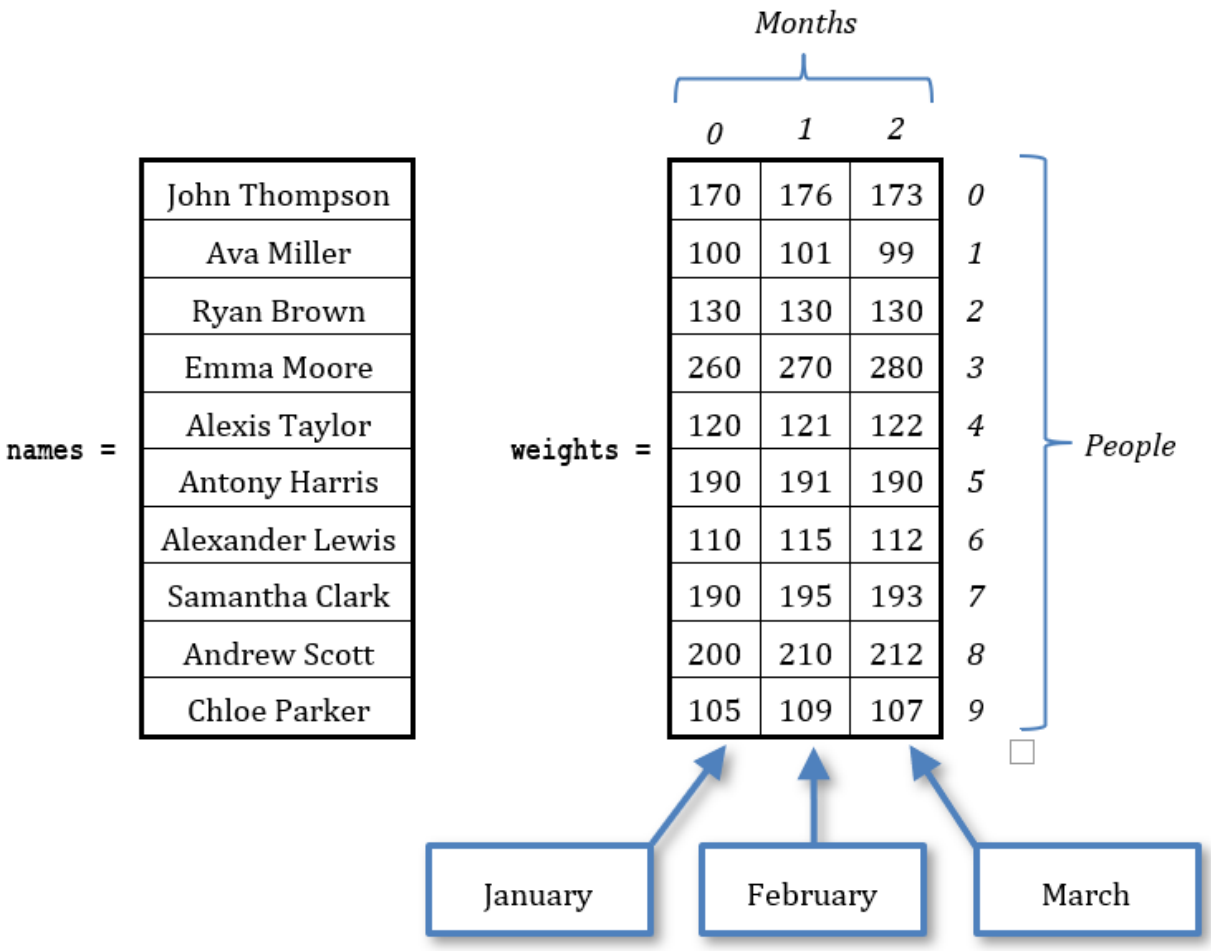
As you can see, there is a one-to-one correspondence between the elements in the array `names` and those in the array `ages`. The first of the seven people is John Thompson, and he is 17 year old. The name "John Thompson" is stored at index 0 of the array `names`, and at exactly the same index in the array `ages`, his age is stored. The next person's name (Ava Miller) and her age (25) are stored at index 1 of the arrays `names`, and `ages`, respectively, and so on.

Exercise 31.2-3 Designing Arrays

Design the necessary arrays to hold the names of ten people as well as the average weight (in pounds) of each person for January, February, and March. Then add some typical values to the arrays.

Solution

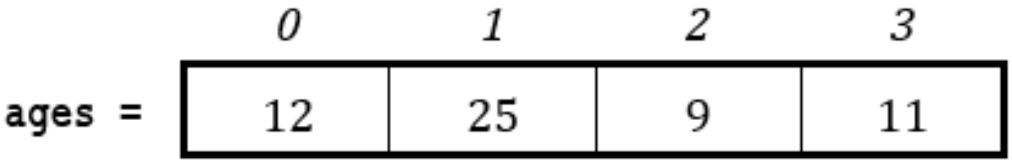
In this exercise, you need a one-dimensional array for names, and a two-dimensional array for people's weights, having a one-to-one correspondence between their elements.



31.3 Creating One-Dimensional Arrays in Java

Java has many ways to create an array and add elements (and values) to it. Depending on the given problem, it's up to you which one to use.

Let's try to create the following array using the most common approaches.



First approach

To create an array and directly assign values to its elements, you can use the next Java statement, given in general form.

```
type[] array_name = { value0, value1, value2, … , valueM };
```

where

- *type* can be `int`, `double`, `String` and so on.

- *array_name* is the name of the array.
- *value0*, *value1*, *value2*, ... , *valueM* are the values of the array elements.

For this approach, you can create the array ages using the following statement:

```
int[] ages = {12, 25, 9, 11};
```

> ✎ *Indexes are set automatically. The value 12 is assigned to the element at index position 0, value 25 is assigned to the element at index position 1, and so on. Index numbering always starts at zero by default.*
>
> ✎ *In Section 5.4 you learned about the rules that must be followed when assigning names to variables. Assigning names to arrays follows exactly the same rules!*

Second approach

You can create an array of *size* empty elements in Java using the following statement given in general form:

```
type[] array_name = new type[size];
```

where *size* can be any positive integer value, or it can even be a variable that contains any positive integer value.

The next statement creates the array ages with 4 empty elements.

```
int[] ages = new int[4];
```

> ✎ *The statement* `int[] ages = new int[4]` *reserves four locations in main memory (RAM).*

To assign a value to an array element, you can use the following statement, given in general form:

```
array_name[index] = value;
```

where *index* is the index position of the element in the array.

The next code fragment creates the array ages (reserving four locations in main memory) and then assigns values to its elements.

```
int[] ages = new int[4];
ages[0] = 12;
ages[1] = 25;
ages[2] = 9;
ages[3] = 11;
```

✎ *The size of the array* ages *is 4.*

Of course, instead of using constant values for *index*, you can also use variables or expressions, as follows.

```
int k;
int[] ages = new int[4];
k = 0;
ages[k] = 12;
ages[k + 1] = 25;
ages[k + 2] = 9;
ages[k + 3] = 11;
```

31.4 How to Get Values from a One-Dimensional Array

Getting values from an array is just a matter of pointing to a specific element. Each element of a one-dimensional array can be uniquely identified using an index. The following code fragment creates an array and displays "A+" (without the double quotes) on the screen.

```
String[] grades = {"B+", "A+", "A", "C-"};
System.out.println(grades[1]);
```

Of course, instead of using constant values for *index*, you can also use variables or expressions. The following example creates an array and displays "Aphrodite and Hera" (without the double quotes) on the screen.

```
String[] gods = {"Zeus", "Ares", "Hera", "Aphrodite", "Hermes"};
k = 2;
System.out.println(gods[k + 1] + " and " + gods[k]);
```

Exercise 31.4-1 Creating the Trace Table

Create the trace table for the next code fragment.

```
int x;
int[] a = new int[4];
a[3] = 9;
x = 0;
a[x] = a[3] + 4;
a[x + 1] = a[x] * 3;
x++;
a[x + 2] = a[x - 1];
a[2] = a[1] + 5;
a[3] = a[3] + 1;
```

Solution

Don't forget that you can manipulate each element of an array as if it were a variable. Thus, when you create a trace table for a Java program that uses arrays, you can have one column for each element as follows.

Step	Statement	Notes	x	a[0]	a[1]	a[2]	a[3]
1	`int[] a = new int[4]`	This creates array a with no values in it	?	?	?	?	?
2	`a[3] = 9`		?	?	?	?	**9**
3	`x = 0`		**0**	?	?	?	9
4	`a[x] = a[3] + 4`		0	**13**	?	?	9
5	`a[x + 1] = a[x] * 3`		0	13	**39**	?	9
6	`x++`		**1**	13	39	?	9
7	`a[x + 2] = a[x - 1]`		1	13	39	?	**13**
8	`a[2] = a[1] + 5`		1	13	39	**44**	13
9	`a[3] = a[3] + 1`		1	13	39	44	**14**

Exercise 31.4-2 Using a Non-Existing Index

Which properties of an algorithm are not satisfied by the following Java program?

```
public static void main(String[] args) throws Exception {
   String[] grades = {"B+", "A+", "A", "C-"};
   System.out.println(grades[100]);
}
```

Solution

Two properties are not satisfied by this Java program. The first one is obvious: there is no data input. The second one is the property of definiteness. You must never reference a non-existing element of an array. In

this exercise, since there is no element at index position 100, the last statement throws a runtime error.

31.5 How to Alter the Value of an Array Element

To alter the value of an existing array element is a piece of cake. All you need to do is use the appropriate index and assign a new value to that element. The example that follows shows exactly this.

```
//Create an array
String[] tribes = {"Navajo", "Cherokee", "Sioux"};
//Alter the value of an existing element
tribes[1] = "Apache";
System.out.println(tribes[0]); //It displays: Navajo
System.out.println(tribes[1]); //It displays: Apache
System.out.println(tribes[2]); //It displays: Sioux
```

✎ *A string is almost identical to an array; it contains a collection of characters. However, the main difference is that strings are immutable (unchangeable). You cannot change an individual character with a statement like* `xStr[2] = "r"` *(although you can change the value of the whole string).*

✎ *An "immutable" data structure is a structure in which the value of its elements cannot be changed once the data structure is created.*

31.6 How to Iterate Through a One-Dimensional Array

Now comes the interesting part. A program can iterate through the elements of an array using a loop control structure (usually a for-loop). There are two approaches you can use to iterate through a one-dimensional array.

First approach

This approach refers to each array element using its index. Following is a code fragment, written in general form

```
for (index = 0; index <= size_of_the_array – 1; index ++) {
    process structure_name[index];
}
```

in which, *process* is any Java statement or block of statements that processes one element of the array *structure_name* at each iteration.

The following code fragment displays all elements of the array `gods`, one at each iteration.

```
int i;
```

```
String[] gods = {"Zeus", "Ares", "Hera", "Aphrodite", "Hermes"};
for (i = 0; i <= 4; i++) {
   System.out.print(gods[i] + "\t");
}
```

✎ *The name of the variable* `i` *is not binding. You can use any variable name you want, such as* `index`, `ind`, `j`, *and many more.*

✎ *Note that since the array* `gods` *contains five elements, the for-loop must iterate from 0 to 4 and not from 1 to 5. This is because the indexes of the four elements are 0, 1, 2, 3, and 4, correspondingly.*

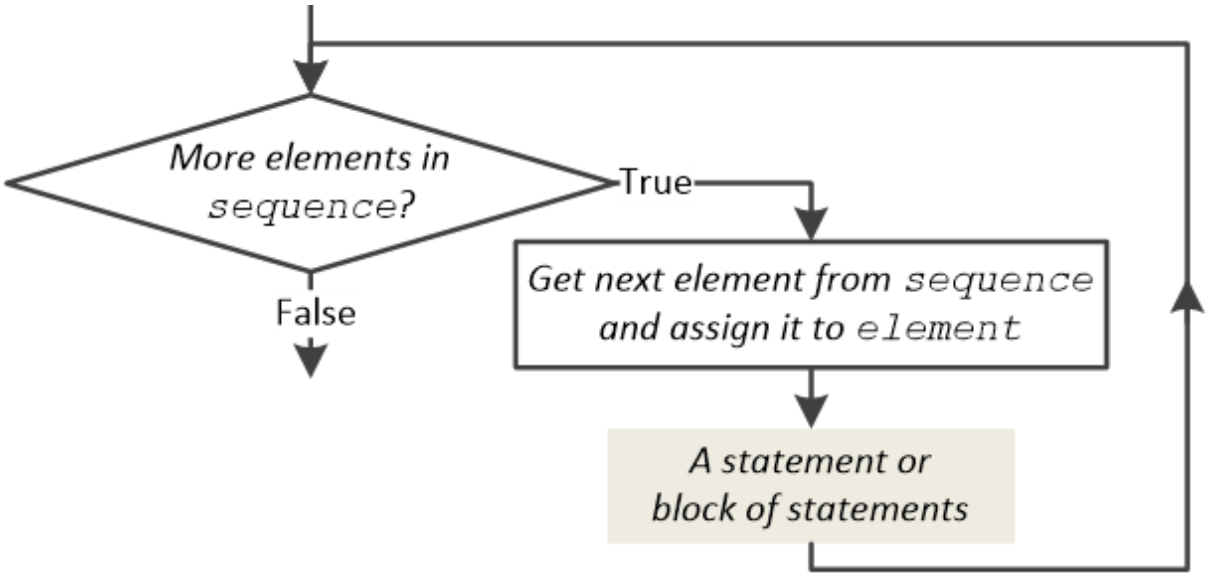
Since arrays are mutable, you can use a loop control structure to alter all or some of its values. The following code fragment doubles the values of some elements of the array `b`.

```
int[] b = {80, 65, 60, 72, 30, 40};
for (i = 0; i <= 3; i++) {
   b[i] = b[i] * 2;
}
```

Second approach

To iterate through the elements of a sequence, apart from the for-loop, there is also another loop, the foreach-loop. Sometimes, it is preferable (or even necessary) to use a foreach-loop.

The general form of the foreach-loop is shown in the following flowchart.


More elements in sequence?
True
False
Get next element from sequence and assign it to element
A statement or block of statements

In Java, the general form of the foreach-loop is

```
for (type element : sequence) {
```

```
    A statement or block of statements
}
```

where

- *element* must be a variable of type *type*. It must match the type of the elements of *sequence*
- *sequence* must be an iterable data structure such as an array.

The following code fragment, displays all elements of the array `grades`, one at each iteration.

```
String[] grades = {"B+", "A+", "A", "C-"};
for (String grade : grades) {
   System.out.println(grade);
}
```

✎ *In the first iteration, the value of the first element is assigned to variable* `grade`*. In the second iteration, the value of the second element is assigned to variable* `grade` *and so on!*

Keep in mind, though, that this approach cannot be used to alter the values of the elements in an array. For example, if you want to double the values of all elements in the array `numbers`, you can**not** do the following:

```
int[] numbers = {5, 10, 3, 2};
for (int number : numbers) {
   number = number * 2;
}
```

✎ `number` *is a simple variable where, at each iteration, each successive value of the array* `numbers` *is assigned to. However, the opposite never happens! The value of* `number` *is never assigned back to any element!*

✎ *If you want to alter the values of the elements in an array, you should use the first approach.*

Exercise 31.6-1 Finding the Sum

Write a Java program that creates an array with the following values

56, 12, 33, 8, 3, 2, 98

and then calculates and displays their sum.

Solution

You learned two approaches to iterate through the array elements. Let's use both approaches and see the differences.

First approach

The solution is as follows.

Project_31.6-1a

```
public static void main(String[] args) throws Exception {
   int i, total;
   int[] values = {56, 12, 33, 8, 3, 2, 98};
   total = 0;
   for (i = 0; i <= 6; i++) {
      total += values[i]; //This is equivalent to total = total + values[i]
   }
   System.out.println(total);
}
```

Second approach

The solution is as follows.

Project_31.6-1b

```
public static void main(String[] args) throws Exception {
   int total;
   int[] values = {56, 12, 33, 8, 3, 2, 98};
   total = 0;
   for (int value : values) {
      total += value;
   }
   System.out.println(total);
}
```

31.7 How to Add User-Entered Values to a One-Dimensional Array

There is nothing new here. Instead of reading a value from the keyboard and assigning that value to a variable, you can directly assign that value to a specific array element. The next code fragment prompts the user to enter the names of four people, and assigns them to the elements at index positions 0, 1, 2, and 3, of the array `names`.

```
public static void main(String[] args) throws Exception {
   String[] names = new String[4]; //Pre-reserve 4 locations in main memory (RAM)
   System.out.print("Enter name No 1: ");
   names[0] = cin.nextLine();
   System.out.print("Enter name No 2: ");
```

```
   names[1] = cin.nextLine();
   System.out.print("Enter name No 3: ");
   names[2] = cin.nextLine();
   System.out.print("Enter name No 4: ");
   names[3] = cin.nextLine();
…
```

Using a for-loop, this code fragment can equivalently be written as

```
static final int ELEMENTS = 4;
public static void main(String[] args) throws Exception {
   int i;
   String[] names = new String[ELEMENTS]; //Pre-reserve 4 locations in main memory (RAM)
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter name No " + (i + 1) + ": ");
      names[i] = cin.nextLine();
   }
…
```

A very good tactic for dealing with array sizes is to use constants.

Exercise 31.7-1 Displaying Words in Reverse Order

Write a Java program that lets the user enter 20 words. The program must then display them in the exact reverse of the order in which they were provided.

Solution

Arrays are perfect for problems like this one. The following is an appropriate solution.

Project_31.7-1

```
public static void main(String[] args) throws Exception {
   int i;
   String[] words = new String[20];
   for (i = 0; i <= 19; i++) {
      words[i] = cin.nextLine();
   }
   for (i = 19; i >= 0; i--) {
      System.out.println(words[i]);
   }
}
```

Since index numbering starts at zero, the index of the last array element is 1 less than the total number of elements in the array.

✎ *Sometimes the wording of an exercise may say nothing about using a data structure. However, this doesn't mean that you can't use one. Use data structures (arrays, hashmaps etc.) whenever you find them necessary.*

Exercise 31.7-2 Displaying Positive Numbers in Reverse Order

Write a Java program that lets the user enter 100 numbers into an array. It then displays only the positive ones in the exact reverse of the order in which they were provided.

Solution

In this exercise, the program must accept all values from the user and store them into an array. However, within the for-loop that is responsible for displaying the array elements, a nested decision control structure must check for and display only the positive values. The solution is as follows.

🗀 **Project_31.7-2**

```
static final int ELEMENTS = 100;
public static void main(String[] args) throws Exception {
   int i;
   double[] values = new double[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      values[i] = Double.parseDouble(cin.nextLine());
   }
   for (i = ELEMENTS - 1; i >= 0; i--) {
      if (values[i] > 0) {
         System.out.println(values[i]);
      }
   }
}
```

Exercise 31.7-3 Finding the Average Value

Write a Java program that prompts the user to enter 20 numbers into an array. It then displays a message only when their average value is less than 10.

Solution

To find the average value of the user-provided numbers the program must first find their sum and then divide that sum by 20. Once the average value is found, the program must check whether to display the corresponding message.

Project_31.7-3a

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
   int i;
   double total, average;
   double[] values = new double[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter a value: ");
      values[i] = Double.parseDouble(cin.nextLine());
   }
   //Accumulate values in total
   total = 0;
   for (i = 0; i <= ELEMENTS - 1; i++) {
      total += values[i];
   }
   average = total / ELEMENTS;
   if (average < 10) {
      System.out.println("Average value is less than 10");
   }
}
```

If you are wondering whether or not this exercise could have been solved using just one for-loop, the answer is “yes”. An alternative solution is presented next.

Project_31.7-3b

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
   int i;
   double total, average;
   total = 0;
   double[] values = new double[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter a value: ");
      values[i] = Double.parseDouble(cin.nextLine());
      total += values[i];
   }
   average = total / ELEMENTS;
   if (average < 10) {
      System.out.println("Average value is less than 10");
   }
}
```

But let's clarify something! Even though many processes can be performed inside just one for-loop, it is simpler to carry out each individual process in a

separate for-loop. This is probably not so efficient but, since you are still a novice programmer, try to adopt this programming style just for now. Later, when you have the experience and become a Java guru, you will be able to "merge" many processes in just one for-loop!

Exercise 31.7-4 Displaying Reals Only

Write a Java program that prompts the user to enter 10 numeric values into an array. The program must then display the indexes of the elements that contain reals.

Solution

In Exercise 23.1-1 you learned how to check whether or not, a number is an integer. Accordingly, to check whether or not, a number is a real (float), you can use the Boolean expression

number `!= (int)`*number*

The solution is as follows.

Project_31.7-4

```
static final int ELEMENTS = 10;
public static void main(String[] args) throws Exception {
   int i;
   double[] a = new double[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter a value for element " + i + ": ");
      a[i] = Double.parseDouble(cin.nextLine());
   }
   for (i = 0; i <= ELEMENTS - 1; i++) {
      if (a[i] != (int)a[i]) {
         System.out.println("A real found at position: " + i);
      }
   }
}
```

Exercise 31.7-5 Displaying Elements with Odd-Numbered Indexes

Write a Java program that prompts the user to enter 8 numeric values into an array. The program must then display the elements with odd-numbered indexes (*that is, indexes 1, 3, 5, and 7*).

Solution

Following is one possible solution.

Project_31.7-5a

```
static final int ELEMENTS = 8;
public static void main(String[] args) throws Exception {
   int i;
   double[] a = new double[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter a value for element " + i + ": ");
      a[i] = Double.parseDouble(cin.nextLine());
   }
   //Display the elements with odd-numbered indexes
   for (i = 0; i <= ELEMENTS - 1; i++) {
      if (i % 2 != 0) {
         System.out.print(a[i] + " ");
      }
   }
}
```

However, you know that only the values in odd-numbered index positions must be displayed. Therefore, the for-loop that is responsible for displaying the elements of the array, instead of starting counting from 0 and using an *offset* of +1, it can start counting from 1 and use an *offset* of +2. This modification decreases the number of iterations by half. The modified Java program follows.

Project_31.7-5b

```
static final int ELEMENTS = 8;
public static void main(String[] args) throws Exception {
   int i;
   double[] a = new double[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter a value for element " + i + ": ");
      a[i] = Double.parseDouble(cin.nextLine());
   }
   //Display the elements with odd-numbered indexes
   for (i = 1; i <= ELEMENTS - 1; i += 2) { //Start from 1 and increment by 2
      System.out.print(a[i] + " ");
   }
}
```

Exercise 31.7-6 Displaying Even Numbers in Odd–Numbered Index Positions

Write a Java program that lets the user enter 100 integers into an array and then displays any even values that are stored in odd–numbered index positions.

Solution

Following is one possible solution.

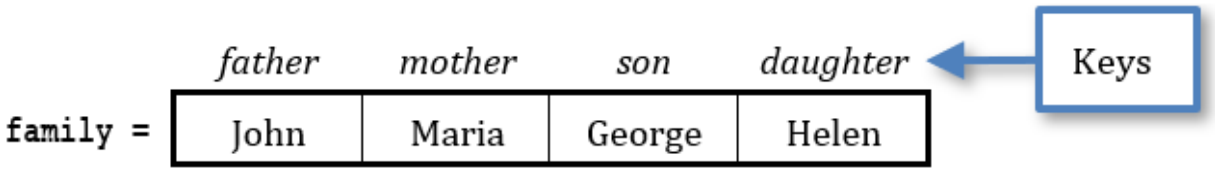
Project_31.7-6

```
static final int ELEMENTS = 100;
public static void main(String[] args) throws Exception {
   int i;
   int[] values = new int[ELEMENTS];
   for (i = 0; i <= ELEMENTS - 1; i++) {
      values[i] = Integer.parseInt(cin.nextLine());
   }
   for (i = 1; i <= ELEMENTS - 1; i += 2) { //Start from 1 and increment by 2
      if (values[i] % 2 == 0) {
         System.out.println(values[i]);
      }
   }
}
```

31.8 What is a HashMap?

In computer science, the main difference between a *hashmap* and an array is that the hashmap elements can be uniquely identified using a key and not necessarily an integer value. Each key of a hashmap is associated (or mapped, if you prefer) to an element. The keys of a hashmap can be of type `String`, `int`, `double` etc.

The following example presents a hashmap that holds the names of a family. The name of the hashmap is `family` and the corresponding keys are written above each element.



✎ *The keys of hashmap elements must be unique within the hashmap. This means that in the hashmap* `family`*, for example, you cannot have two keys named* `father`.

✎ *The values of hashmap elements can be of any type.*

31.9 Creating HashMaps in Java

Let's try to create the following hashmap using the most common approaches.

	firstName	*lastName*	*age*	*class*
pupil =	Ann	Fox	8	2nd

First approach

To create a hashmap and directly assign values to its elements, you can use the next Java statement, given in general form.

```
HashMap<key_type, value_type> hashmap_name = new HashMap<>(
   Map.of(key0, value0, key1, value1, key2, value2, … keyM, valueM)
);
```

where

- *key_type* is the type of the keys. It can be `String`, `Integer`, `Double` etc.
- *value_type* is the type of the elements. It can be `String`, `Integer`, `Double` etc.
- *hashmap_name* is the name of the hashmap.
- *key0*, *key1*, *key2*, ... , *keyM* are the keys of the hashmap elements.
- *value0*, *value1*, *value2*, ... , *valueM* are the values of the hashmap elements.

Using this approach, the hashmap `pupil` can be created using the following statement:

```
HashMap<String, String> pupil = new HashMap<>(
   Map.of("firstName", "Ann", "lastName", "Fox", "age", "8", "class", "2nd")
);
```

✎ *The* `Map.of()` *method was first introduced in Java 9.0. It supports up to 10 key-value pairs.*

✎ *In Section 5.4 you learned about the rules that must be followed when assigning names to variables. Assigning names to hashmaps follows exactly the same rules!*

✎ *In order to use the* `HashMap` *and* `Map` *classes in your project, you need to import the corresponding libraries using the* `import java.util.HashMap` *and* `import java.util.Map` *statements at the beginning of the project. You will learn more about classes and objects in Part VIII.*

Second approach

In this approach, you can create a totally empty hashmap using the following statement, given in general form

```
HashMap<key_type, value_type> hashmap_name = new HashMap<>();
```

and then add an element (key-value pair), as shown in the following Java statement, given in general form.

```
hashmap_name.put(key, value);
```

Using this approach, the hashmap `pupil` can be created using the following code fragment:

```
HashMap<String, String> pupil = new HashMap<>();
pupil.put("firstName", "Ann");
pupil.put("lastName", "Fox");
pupil.put("age", "8");
pupil.put("class", "2nd");
```

31.10 How to Get a Value from a HashMap

To get the value of a specific hashmap element, you must point to that element using its corresponding key. The following code fragment creates a hashmap, and then displays “Ares is the God of War”, without the double quotes, on the screen.

```
HashMap<String, String> olympians = new HashMap<>();
olympians.put("Zeus", "King of the Gods");
olympians.put("Hera", "Goddess of Marriage");
olympians.put("Ares", "God of War");
olympians.put("Poseidon", "God of the Sea");
olympians.put("Demeter", "Goddess of the Harvest");
olympians.put("Artemis", "Goddess of the Hunt");
olympians.put("Apollo", "God of Music and Medicine");
olympians.put("Aphrodite", "Goddess of Love and Beauty");
olympians.put("Hermes", "Messenger of the Gods");
olympians.put("Athena", "Goddess of Wisdom");
olympians.put("Hephaistos", "God of Fire and the Forge");
olympians.put("Dionysus", "God of the Wine");
System.out.println("Ares is the " + olympians.get("Ares"));
```

✎ *Only keys can be used to access an element. This means that* `olympians.get("Ares")` *correctly returns “God of War” but* `olympians.get("God of War")` *cannot return “Ares”.*

Exercise 31.10-1 Roman Numerals to Numbers

Roman numerals are shown in the following table.

Number	Roman Numeral
1	I
2	II
3	III
4	IV
5	V

Write a Java program that prompts the user to enter a Roman numeral between I and V, and then displays the corresponding number. Assume that the user enters a valid value.

Solution

The obvious solution would be the use of a multiple-alternative decision structure, similar to the one shown in the code fragment that follows.

```
if (romanNumeral == "I")
   number = 1;
else if (romanNumeral == "II")
   number = 2;
else if (romanNumeral == "III")
   number = 3;
else if (romanNumeral == "IV")
   number = 4;
else if (romanNumeral == "V")
   number = 5;
```

However, this approach is quite lengthy, and it could become even more extensive if you want to expand your program to work with additional Roman numerals. Therefore, armed with knowledge about hashmaps, you can employ a more efficient approach, as demonstrated in the code fragment that follows.

```
HashMap<String, Integer> roman2number = new HashMap<>(
   Map.of("I", 1, "II", 2, "III", 3, "IV", 4, "V", 5)
);
number = roman2number[romanNumeral];
```

The solution to this exercise is as follows.

Project_31.10-1

```
import java.util.Scanner;
import java.util.Map;
```

```
import java.util.HashMap;
public class App {
   static Scanner cin = new Scanner(System.in);
   public static void main(String[] args) throws Exception {
      String romanNumeral;
      int number;
      HashMap<String, Integer> roman2number = new HashMap<>(
         Map.of("I", 1, "II", 2, "III", 3, "IV", 4, "V", 5)
      );
      System.out.print("Enter a Roman numeral: ");
      romanNumeral = cin.nextLine();
      number = roman2number.get(romanNumeral);
      System.out.println(romanNumeral + ": " + number);
   }
}
```

In order to use the `HashMap` *and* `Map` *classes in your project, you need to import the corresponding libraries using the* `import java.util.HashMap` *and* `import java.util.Map` *statements at the beginning of the project*

Exercise 31.10-2 Using a Non-Existing Key in HashMaps

What is wrong in the following Java program?

```
HashMap<String, String> family = new HashMap<>(
   Map.of("father", "John", "mother", "Maria", "son", "George")
);
System.out.println(family.get("daughter"));
```

Solution

Similar to arrays, this code does not satisfy the property of definiteness. You must never reference a non-existing hashmap element. Since there is no key “daughter”, the last statement displays “null”.

In computer science, “null” means “nothing”. When a variable or a data structure element has no value, it is considered to be null. A value of 0 is different than the null value, since 0 is an actual value. Regarding strings, the empty string ("") is also different than the null value. An empty string is a string with no characters in it whereas a null string has no value at all!

31.11 How to Alter the Value of a HashMap Element

To alter the value of an existing hashmap element you need to use the appropriate key and assign a new value to that element. The example that

follows shows exactly this.

```
HashMap<String, String> tribes = new HashMap<>(
   Map.of("Indian", "Navajo", "African", "Zulu")
);
System.out.println(tribes); //It displays: {Indian=Navajo, African=Zulu}
//Alter the value of an existing element
tribes.put("Indian", "Apache");
System.out.println(tribes); //It displays: {Indian=Apache, African=Zulu}
```

Exercise 31.11-1 Assigning a Value to a Non-Existing Key

Is there anything wrong in the following code fragment?

```
HashMap<Integer, String> tribes = new HashMap<>(
   Map.of(0, "Navajo", 1, "Cherokee", 2, "Sioux")
);
tribes.put(3, "Apache");
```

Solution

No, this time there is absolutely nothing wrong in this code fragment. At first glance, you might have thought that the last statement tries to alter the value of a non-existing key and it will throw an error. This is **not** true for Java's hashmaps, though. Since `tribes` is a hashmap and key “3” does not exist, the last statement **adds** a brand new fourth element to the hashmap!

The keys of a hashmap can be of type `String`, `int`, `double` *etc.*

Keep in mind though, if `tribes` were actually an array, the last statement would certainly throw an error. Take a look at the following code fragment

```
String[] tribes = {"Navajo", "Cherokee", "Sioux"};
tribes[3] = "Apache";
```

In this example, since `tribes` is an array and index 3 does not exist, the last statement tries to **alter** the value of a non-existing element and obviously throws an error!

31.12 How to Iterate Through a HashMap

To iterate through the elements of a hashmap you can use a for-loop. Following is a code fragment, written in general form

```
for (key_type key : structure_name.keySet()) {
    process structure_name.get(key);
}
```

in which *process* is any Java statement or block of statements that processes one element of the hashmap *structure_name* at each iteration.

The following Java program displays the letters A, B, C, and D, and their corresponding Morse[23] code.

```
HashMap<String, String> morseCode = new HashMap<>(
   Map.of("A", ".-", "B", "-...", "C", "-.-.", "D", "-..")
);
for (String letter : morseCode.keySet()) {
   System.out.println(letter + " " + morseCode.get(letter));
}
```

The next example gives a bonus of $2000 to each employee of a computer software company!

```
HashMap<String, Double> salaries = new HashMap<>(
   Map.of("Project Manager", 83000.0,
            "Software Engineer", 81000.0,
            "Network Engineer", 64000.0,
            "Systems Administrator", 61000.0,
            "Software Developer", 70000.0
   )
);
for (String title : salaries.keySet()) {
   salaries.put(title, salaries.get(title) + 2000);
}
```

31.13 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Arrays are structures that can hold multiple values.
2) Array elements are located in main memory (RAM).
3) There can be only one-dimensional and two-dimensional arrays.
4) There cannot be four-dimensional arrays.
5) An array is called “multidimensional” because it can hold values of different types.
6) Each array element has a unique non-negative index.
7) There can be two identical keys within a hashmap.
8) In arrays, index numbering always starts at zero by default.
9) The index of the last array element is equal to the total number of its elements.

10) A two-hundred–dimensional array can exist.
11) The next statement contains a syntax error.

```
String[] studentNames = String[10];
```

12) In a Java program, two arrays cannot have the same name.
13) The next statement is syntactically correct.

```
HashMap<String, String> student = new HashMap<>(
   Map.of("firstName": "Ann", "lastName": "Fox", "age": "8")
);
```

14) In a Java program, two arrays cannot have the same number of elements.
15) You cannot use a variable as an index in an array.
16) You can use a mathematical expression as an index in an array.
17) You cannot use a variable as a key in a hashmap.
18) The following code fragment throws no errors.

```
String a = "a";
HashMap<String, String> fruits = new HashMap<>(
   Map.of("o", "Orange", "a", "Apple", "w", "Watermelon")
);
System.out.println(fruits.get(a));
```

19) If you use a variable as an index in an array, this variable must contain an integer value.
20) In order to calculate the sum of 20 numeric user-provided values, you must use an array.
21) You can let the user enter a value into array `b` using the statement `b[k] = cin.nextLine()`
22) The following statement creates a one-dimensional array of two empty elements.

```
String[] names = new String[3];
```

23) The following code fragment assigns the value 10 to the element at index 7.

```
values[5] = 7;
values[values[5]] = 10;
```

24) The following code fragment assigns the value “Sally” without the double quotes to the element at index 2.

```
String[] names = new String[3];
```

```
names[2] = "John";
names[1] = "George";
names[0] = "Sally";
```

25) The following statement assigns the value “Sally” without the double quotes to the element at index 2.

```
String[] names = {"John", "George", "Sally"};
```

26) The following code fragment displays “Sally”, without the double quotes, on the screen.

```
String[] names = new String[3];
k = 0;
names[k] = "John";
k++;
names[k] = "George";
k++;
names[k] = "Sally";
k--;
System.out.println(names[k]);
```

27) The following code fragment is syntactically correct.

```
String[] names = {"John", "George", "Sally"};
System.out.println(names[]);
```

28) The following code fragment displays “Maria”, without the double quotes, on the screen.

```
String[] names = {"John", "George", "Sally", "Maria"};
System.out.println(names[(int)Math.PI]);
```

29) The following code fragment satisfies the property of definiteness.

```
String[] grades = {"B+", "A+", "A"};
System.out.println(grades[3]);
```

30) The following code fragment satisfies the property of definiteness.

```
int[] v = {1, 3, 2, 9};
System.out.println(v[v[v[0]]]);
```

31) The following code fragment displays the value of 1 on the screen.

```
int[] v = {1, 3, 2, 0};
System.out.println(v[v[v[v[0]]]]);
```

32) The following code fragment displays all the elements of the array `names`.

```
String[] names = {"John", "George", "Sally", "Maria"};
i = 1;
while (i < 4) {
   System.out.println(names[i++]);
```

```
}
```

33) The following code fragment satisfies the property of definiteness.

```
String[] names = {"John", "George", "Sally", "Maria"};
for (i = 2; i <= 4; i++) {
   System.out.println(names[i]);
}
```

34) The following code fragment lets the user enter 100 values into array `b`.

```
for (i = 0; i <= 99; i++) {
   b[i] = Integer.parseInt(cin.nextLine());
}
```

35) If array `b` contains 30 elements (arithmetic values), the following code fragment doubles the values of all of its elements.

```
for (i = 29; i >= 0; i--) {
   b[i] = b[i] * 2;
}
```

36) It is possible to use a for-loop to double the values of some of the elements of an array.

37) If array `b` contains 30 elements, the following code fragment displays all of them.

```
for (i = 0; i < 29; i++) {
   System.out.println(b[i]);
}
```

38) If `b` is a hashmap, the following code fragment displays all of its elements.

```
for (int id : b.keySet()) {
   System.out.println(b.get());
}
```

39) The following code fragment throws an error.

```
HashMap<String, String> fruits = new HashMap<>(
   Map.of("O", "Orange", "A", "Apple", "W", "Watermelon")
);
System.out.println(fruits.get("Orange"));
```

31.14 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The following statement

```
String() lastNames = new String[4];
```

   a) contains a logic error.

b) contains a syntax error.
c) is a correct statement.
d) none of the above

2) The following code fragment

```
double x = 5;
values[x / 2] = 10;
```

a) does not satisfy the property of definiteness.
b) does not satisfy the property of finiteness.
c) does not satisfy the property of effectiveness.
d) none of the above

3) If variable x contains the value 4, the following statement

```
values[x + 1] = 5;
```

a) assigns the value 4 to the element at index 5.
b) assigns the value 5 to the element at index 4.
c) assigns the value 5 to the element at index 5.
d) none of the above

4) The following statement

```
int[] values = {5, 6, 9, 1, 1, 1};
```

a) assigns the value 5 to the element at index 1.
b) assigns the value 5 to the element at index 0.
c) does not satisfy the property of definiteness.
d) none of the above

5) The following code fragment

```
values[0] = 1;
values[values[0]] = 2;
values[values[1]] = 3;
values[values[2]] = 4;
```

a) assigns the value 4 to the element at index 3.
b) assigns the value 3 to the element at index 2.
c) assigns the value 2 to the element at index 1.
d) all of the above
e) none of the above

6) If array `values` contains numeric values, the following statement

```
System.out.println(values[values[1] - values[1 % 2]] - values[(int)(1 / 2)]);
```

   a) does not satisfy the property of definiteness.
   b) always displays 0.
   c) always displays 1.
   d) none of the above

7) You can iterate through a one-dimensional array with a for-loop that uses
   a) variable `i` as a counter.
   b) variable `j` as a counter.
   c) variable `k` as a counter.
   d) any variable as a counter.

8) The following code fragment

```
String[] names = {"George", "John", "Maria", "Sally"};
for (i = 3; i >= 1; i--) {
   System.out.println(names[i]);
}
```

   a) displays all names in ascending order.
   b) displays some names in ascending order.
   c) displays all names in descending order.
   d) displays some names in descending order.
   e) none of the above

9) The following code fragment

```
String[] fruits = {"apple", "orange", "onion", "watermelon"};
System.out.println(fruits[1]);
```

   a) displays: "orange"
   b) displays: apple
   c) displays: orange
   d) throws an error because onion is not a fruit!
   e) none of the above

10) If array `b` contains 30 elements (arithmetic values), the following code fragment

```
for (i = 29; i >= 1; i--) {
```

```
   b[i] = b[i] * 2;
}
```

a) doubles the values of some of its elements.

b) doubles the values of all of its elements.

c) none of the above

11) The following code fragment

```
HashMap<String, String> struct = new HashMap<>(
   Map.of("firstName", "George", "lastName", "Miles", "age", "28")
);
for (String a : struct.keySet()) {
   System.out.println(struct.get(a));
}
```

a) displays all the keys of the hashmap elements.

b) displays all the values of the hashmap elements.

c) displays all the key-value pairs of the hashmap elements.

d) none of the above

12) The following code fragment

```
HashMap<String, String> struct = new HashMap<>(
   Map.of("firstName", "George", "lastName", "Miles", "age", "28")
);
for (String x : struct.keySet()) {
   System.out.println(x);
}
```

a) displays all the keys of the hashmap elements.

b) displays all the values of the hashmap elements.

c) displays all the key-value pairs of the hashmap elements.

d) none of the above

13) The following code fragment

```
HashMap<Integer, String> tribes = new HashMap<>(
   Map.of(0, "Navajo", 1, "Cherokee", 2, "Sioux", 3, "Apache")
);
for (i = 0; i <= 3; i++) {
   System.out.println(tribes.get(i));
}
```

a) displays all the keys of the hashmap elements.

b) displays all the values of the hashmap elements.

c) displays all the key-value pairs of the hashmap elements.

d) none of the above

14) The following code fragment

```
HashMap<String, String> tribes = new HashMap<>(
   Map.of("tribeA", "Navajo", "tribeB", "Cherokee", "tribeC", "Sioux")
);
for (String x : tribes.keySet()) {
   tribes.put(x, tribes.get(x).toUpperCase());
}
```

a) converts all the keys of the hashmap elements to uppercase.

b) converts all the values of the hashmap elements to uppercase.

c) convert all the key-value pairs of the hashmap elements to uppercase.

d) none of the above

31.15 Review Exercises

Complete the following exercises.

1) Design a data structure to hold the weights (in pounds) of five people, and then add some typical values to the structure.
2) Design the necessary data structures to hold the names and the weights (in pounds) of seven people, and then add some typical values to the structures.
3) Design the necessary data structures to hold the names of five lakes as well as the average area (in square miles) of each lake in June, July, and August. Then add some typical values to the structures.
4) Design a data structure to hold the three dimensions (width, height, and depth in inches) of 10 boxes. Then add some typical values to the structure.
5) Design the necessary data structures to hold the names of eight lakes as well as the average area (in square miles) and maximum depth (in feet) of each lake. Then add some typical values to the structures.
6) Design the necessary data structures to hold the names of four lakes as well as their average areas (in square miles) for the first week of June, the first week of July, and the first week of August.
7) Create the trace table for the following code fragment.

```
int[] a = new int[3];
a[2] = 1;
x = 0;
a[x + a[2]] = 4;
a[x] = a[x + 1] * 4;
```

8) Create the trace table for the following code fragment.

```
int[] a = new int[5];
a[1] = 5;
x = 0;
a[x] = 4;
a[a[0]] = a[x + 1] % 3;
a[a[0] / 2] = 10;
x += 2;
a[x + 1] = a[x] + 9;
```

9) Create the trace table for the following code fragment for three different executions.

   The input values for the three executions are: (i) 3, (ii) 4, and (iii) 1.

```
int[] a = new int[4];
a[1] = Integer.parseInt(cin.nextLine());
x = 0;
a[x] = 3;
a[a[0]] = a[x + 1] % 2;
a[a[0] % 2] = 10;
x++;
a[x + 1] = a[x] + 9;
```

10) Create the trace table for the following code fragment for three different executions.

    The input values for the three executions are: (i) 100, (ii) 108, and (iii) 1.

```
int[] a = new int[4];
a[1] = Integer.parseInt(cin.nextLine());
x = 0;
a[x] = 3;
a[a[0]] = a[x + 1] % 10;
if (a[3] > 5) {
   a[a[0] % 2] = 9;
   x += 1;
   a[x + 1] = a[x] + 9;
}
else {
   a[2] = 3;
}
```

11) Fill in the gaps in the following trace table. In steps 6 and 7, fill in the name of a variable; for all other cases, fill in constant values, arithmetic, or comparison operators.

Step	Statement	x	y	a[0]	a[1]	a[2]
1	`int[] a = new int[3]`	?	?	?	?	?
2	`x = ……`	**4**	?	?	?	?
3	`y = x - ……`	4	**3**	?	?	?

4, 5	`if (x …… y)`
	`a[0] = …… ;`
	`else`
	`a[0] = y;`
		4	3	**1**	?	?

6	`a[1] = …… + 3`	4	3	1	**7**	?
7	`y = …… - 1`	4	**2**	1	7	?
8	`a[y] = (x + 5) …… 2`	4	2	1	7	**1**

12) Create the trace table for the following code fragment.

```
int[] a = {17, 12, 45, 12, 12, 49};
for (i = 0; i <= 5; i++) {
   if (a[i] == 12)
      a[i]--;
   else
      a[i]++;
}
```

13) Create the trace table for the following code fragment.

```
int[] a = {10, 15, 12, 23, 22, 19};
for (i = 1; i <= 4; i++) {
   a[i] = a[i + 1] + a[i - 1];
}
```

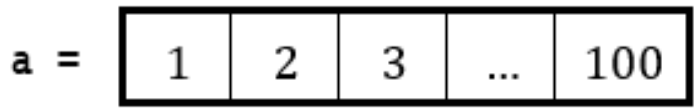
14) Try, without using a trace table, to determine the values that are displayed when the following code fragment is executed.

```
HashMap<String, String> tribes = new HashMap<>(
   Map.of("Indian-1", "Navajo","Indian-2", "Cherokee", "Indian-3", "Sioux",
            "African-1", "Zulu", "African-2", "Maasai", "African-3", "Yoruba")
);
for (String x : tribes.keySet()) {
   if (x.substring(0, 6).equals("Indian")) {
```

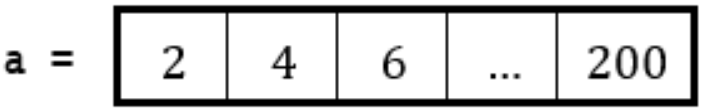
```
        System.out.println(tribes.get(x));
    }
}
```

15) Write a Java program that lets the user enter 100 numbers into an array and then displays these values raised to the power of three.
16) Write a Java program that lets the user enter 80 numbers into an array. Then, the program must raise the array values to the power of two, and finally display them in the exact reverse of the order in which they were provided.
17) Write a Java program that lets the user enter 90 integers into an array and then displays those that are exactly divisible by 5 in the exact reverse of the order in which they were provided.
18) Write a Java program that lets the user enter 50 integers into an array and then displays those that are even or greater than 10.
19) Write a Java program that lets the user enter 30 numbers into an array and then calculates and displays the sum of those that are positive.
20) Write a Java program that lets the user enter 50 integers into an array and then calculates and displays the sum of those that have two digits.

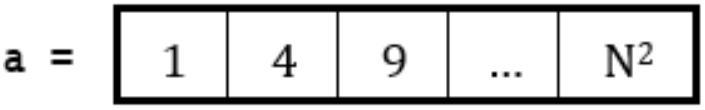
    Hint: All two-digit integers are between 10 and 99.
21) Write a Java program that lets the user enter 40 numbers into an array and then calculates and displays the sum of the positive numbers and the sum of the negative ones.
22) Write a Java program that lets the user enter 20 numbers into an array and then calculates and displays their average value.
23) Write a Java program that prompts the user to enter 50 integer values into an array. It then displays the indexes of the elements that contain values lower than 20.
24) Write a Java program that prompts the user to enter 60 numeric values into an array. It then displays the elements with even-numbered indexes (that is, indexes 0, 2, 4, 6, and so on).
25) Write a Java program that prompts the user to enter 20 numeric values into an array. It then calculates and displays the sum of the elements that have even indexes.
26) Write a code fragment in Java that creates the following array of 100 elements.



27) Write a code fragment in Java that creates the following array of 100 elements.



28) Write a Java program that prompts the user to enter an integer `N` and then creates and displays the following array of `N` elements. Using a loop control structure, the program must also validate data input and display an error message when the user enters any value less than 1.



29) Write a Java program that prompts the user to enter 10 numeric values into an array and then displays the indexes of the elements that contain integers.

30) Write a Java program that prompts the user to enter 50 numeric values into an array and then counts and displays the total number of negative elements.

31) Write a Java program that prompts the user to enter 50 words into an array and then displays those that contain at least 10 characters.

    Hint: Use the `length()` method.

32) Write a Java program that lets the user enter 30 words into an array. It then displays those words that have less than 5 characters, then those that have less than 10 characters, and finally those that have less than 20 characters.

    Hint: Try to display the words using two for-loops nested one within the other.

33) Write a Java program that prompts the user to enter 40 words into an array and then displays those that contain the letter “w” at least twice.

34) Roman numerals are shown in the following table.

Digit	Roman Numeral (Tens digits)	Roman Numeral (Ones digits)
1	X	I

2	XX	I
3	XXX	III
4	XL	IV
5	L	V
6	LX	VI
7	LXX	VII
8	LXXX	VIII
9	XC	IX

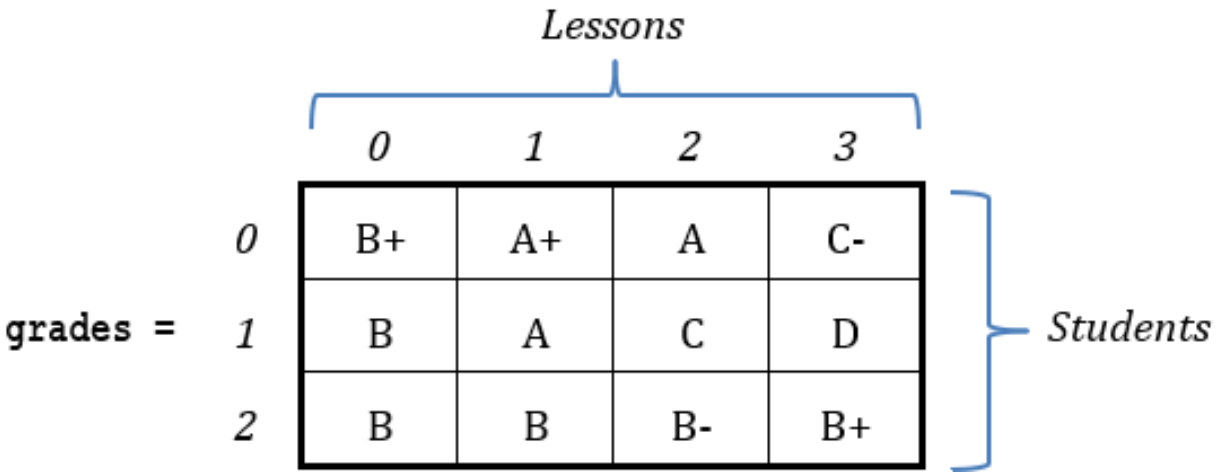
Each Roman numeral is a combination of tens and ones digits. For example, the Roman numeral representation of the number 45 is XLV (4 tens represented by XL and 5 ones represented by V). Write a Java program that prompts the user to enter an integer between 1 and 99 and displays the corresponding Roman numeral. Assume that the user enters a valid value.

Hint: Avoid checking each integer individually, as this would require a multiple-alternative decision structure with 99 cases. Try to find a more efficient and clever approach instead!

Chapter 32
Two-Dimensional Arrays

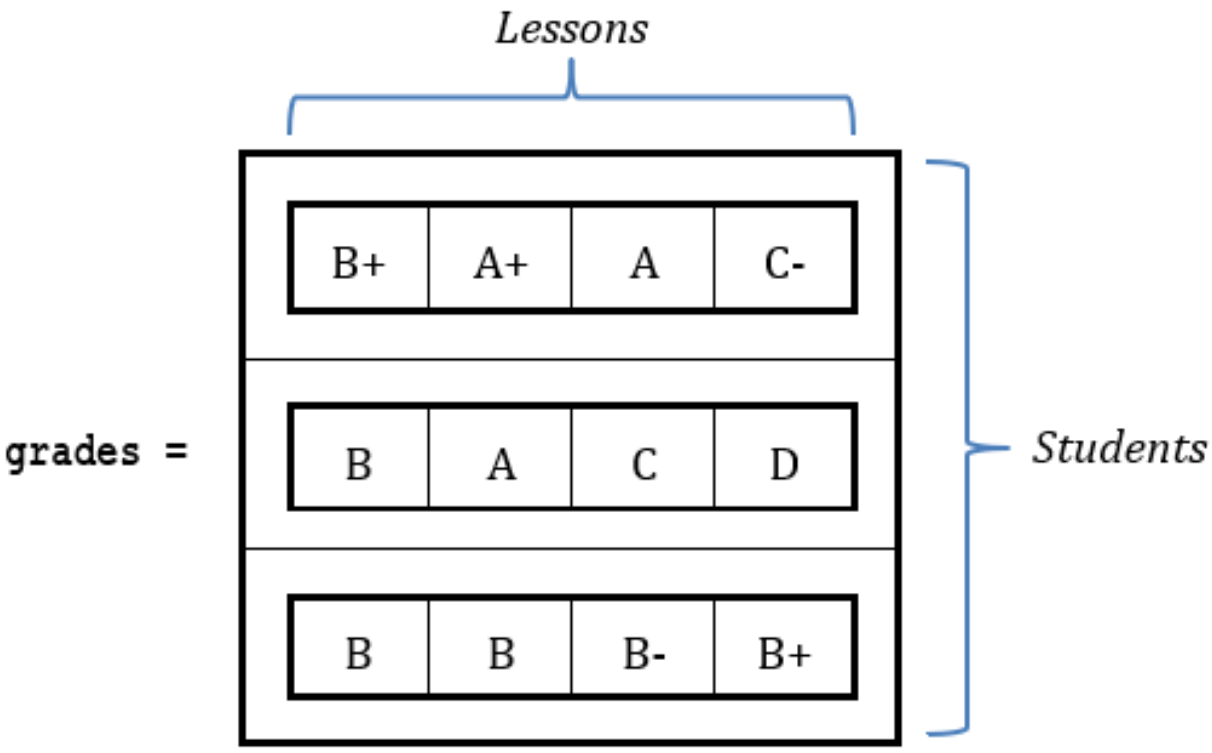
32.1 Creating Two-Dimensional Arrays in Java

A two-dimensional array is a data structure that can store values organized in rows and columns. It allows you to efficiently represent and manipulate tabular data. For instance, an array that can hold the grades of four lessons for three students is as follows.



✎ *A two-dimensional array has rows and columns. In this particular example, array* `grades` *has 3 rows and 4 columns.*

Unlike some other programming languages, Java does not directly support two-dimensional arrays. Java primarily supports only one-dimensional arrays, but there is a trick you can use to work around this and create multidimensional arrays: you can create an array of arrays! Think of the `grades` array as a single-column array with three elements (one for each student), each of which contains a completely new array of four elements (one for each lesson), as follows.



✎ *In Java, a two-dimensional array is an array of arrays, a three-dimensional array is an array of arrays of arrays, and so on.*

As in one-dimensional arrays, there are many ways to create and add elements (and values) to a two-dimensional array. Let's try to create the array `grades` using the most common approaches.

First approach

To create an array and directly assign values to its elements, you can use the next Java statement, given in general form.

```
type[][] array_name = {
   {value0-0, value0-1, value0-2, …, value0-M},
   {value1-0, value1-1, value1-2, …, value1-M},
   {value2-0, value2-1, value2-2, …, value2-M},
   …
   {valueN-0, valueN-1, valueN-2, …, valueN-M}
};
```

where

- *type* can be `int`, `double`, `String` and so on.
- *array_name* is the name of the array.
- *value0-0*, *value0-1*, *value0-2*, …, *valueN-M* are the values of the array elements.

For this approach, you can create the array `grades` using the following statement:

```
String[][] grades = {
   {"B+", "A+", "A", "C-"},
   {"B", "A", "C", "A+"},
   {"B", "B", "B-", "B+"}
};
```

which can also be written in one line as

```
String[][] grades = {{"B+", "A+", "A", "C-"}, {"B", "A", "C", "A+"}, {"B", "B", "B-",
"B+"}};
```

> ✎ *Indexes are set automatically. The first value "B+" is assigned to the element at row index 0 and column index 0, second value "A+" is assigned to the element at row index 0 and column index 1, and so on.*

Second approach

You can create a two-dimensional array with empty elements in Java using the following statement, given in general form,

```
type[][] array_name = new type[number_of_rows][number_of_columns];
```

where

- *type* can be `int`, `double`, `String` and so on.
- *array_name* is the name of the array.
- *number_of_rows* and *number_of_columns* can be any positive integer value.

Then you can assign a value to an array element using the following statement, given in general form:

```
array_name[row_index][column_index] = value;
```

where *row_index* and *column_index* are the row index and the column index positions, respectively, of the element in the array.

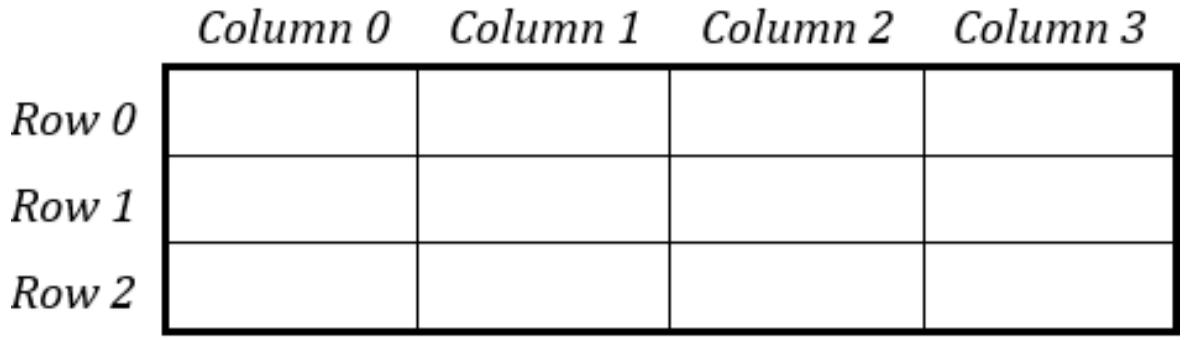
The following code fragment creates the array `grades` with 12 empty elements arranged in 3 rows and 4 columns and then assigns values to its elements.

```
String[][] grades = new String[3][4];
grades[0][0] = "B+";
grades[0][1] = "A+";
grades[0][2] = "A";
grades[0][3] = "C-";
grades[1][0] = "B";
grades[1][1] = "A";
```

```
grades[1][2] = "C";
grades[1][3] = "A+";
grades[2][0] = "B";
grades[2][1] = "B";
grades[2][2] = "B-";
grades[2][3] = "B+";
```

32.2 How to Get Values from Two-Dimensional Arrays

A two-dimensional array consists of rows and columns. The following example shows a two-dimensional array with three rows and four columns.



Each element of a two-dimensional array can be uniquely identified using a pair of indexes: a row index, and a column index, as shown next.

```
array_name[row_index][column_index]
```

The following code fragment creates the two-dimensional array `grades` having three rows and four columns, and then displays some of its elements.

```
String[][] grades = {
   {"B+", "A+", "A", "C-"},
   {"B", "A", "C", "D"},
   {"B", "B", "B-", "B+"}
};
System.out.println(grades[1][2]); //It displays: C
System.out.println(grades[2][2]); //It displays: B-
System.out.println(grades[0][0]); //It displays: B+
```

Exercise 32.2-1 Creating the Trace Table

Create the trace table for the next code fragment.

```
int[][] a = {
   {0, 0},
   {0, 0},
   {0, 0}
};
a[1][0] = 9;
a[0][1] = 1;
```

```
a[0][0] = a[0][1] + 6;
x = 2;
a[x][1] = a[0][0] + 4;
a[x - 1][1] = a[0][1] * 3;
a[x][0] = a[x - 1][1] - 3;
```

Solution

This code fragment uses a 3 × 2 array, that is, an array that has 3 rows and 2 columns. The trace table is as follows.

Step	Statement	Notes	x	a
1	`int[][] a = { {0, 0}, {0, 0}, {0, 0} }`	This creates the array a with zero values in it.	?	**0** **0** **0** **0** **0** **0**
2	`a[1][0] = 9`		?	0 0 **9** 0 0 0
3	`a[0][1] = 1`		?	0 **1** 9 0 0 0
4	`a[0][0] = a[0][1] + 6`		?	**7** 1 9 0 0 0
5	`x = 2`		**2**	7 1 9 0 0 0
6	`a[x][1] = a[0][0] + 4`		2	7 1

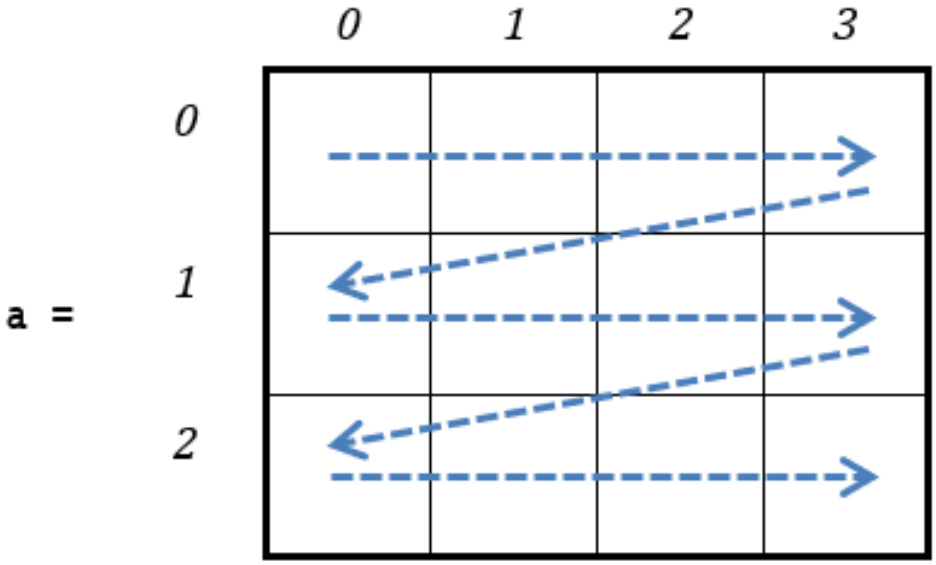
				9 0 0 **11**
7	`a[x - 1][1] = a[0][1] * 3`		2	7 1 9 **3** 0 11
8	`a[x][0] = a[x - 1][1] - 3`		2	7 1 9 3 **0** 11

32.3 How to Iterate Through a Two-Dimensional Array

Since a two-dimensional array consists of rows and columns, a program can iterate either through rows or through columns.

Iterating through rows

Iterating through rows means that row 0 is processed first, row 1 is process next, row 2 afterwards, and so on. Next there is an example of a 3 × 4 array. The arrows show the "path" that is followed when iteration through rows is performed or in other words, they show the order in which the elements are processed.



A 3 × 4 array is a two-dimensional array that has 3 rows and 4 columns. In the notation Y × X, the first number (Y) always represents the total number of rows and the second number (X) always represents the total number of columns.

When iterating through rows, the elements of the array are processed as follows:

- the elements of row 0 are processed in the following order

  `a[0][0]` → `a[0][1]` → `a[0][2]` → `a[0][3]`
- the elements of row 1 are processed in the following order

  `a[1][0]` → `a[1][1]` → `a[1][2]` → `a[1][3]`
- the elements of row 2 are processed in the following order

  `a[2][0]` → `a[2][1]` → `a[2][2]` → `a[2][3]`

First approach

Using Java statements, let's try to process all elements of a 3 × 4 array (3 rows × 4 columns) iterating through rows.

```
i = 0; //Variable i refers to row 0.
for (j = 0; j <= 3; j++) { //This loop control structure processes all elements of row 0
    process a[i][j];
}
i = 1; //Variable i refers to row 1.
for (j = 0; j <= 3; j++) { //This loop control structure processes all elements of row 1
    process a[i][j];
}
i = 2; //Variable i refers to row 2.
for (j = 0; j <= 3; j++) { //This loop control structure processes all elements of row 2
    process a[i][j];
}
```

Of course, the same results can be achieved using a nested loop control structure as shown next.

```
for (i = 0; i <= 2; i++) {
   for (j = 0; j <= 3; j++) {
       process a[i][j];
   }
}
```

Let's see some examples. The following code fragment lets the user enter 10 × 10 = 100 values into array `b`.

```
for (i = 0; i <= 9; i++) {
   for (j = 0; j <= 9; j++) {
      b[i][j] = cin.nextLine();
   }
}
```

The following code fragment decreases all values of array `b` by one.

```
for (i = 0; i <= 9; i++) {
   for (j = 0; j <= 9; j++) {
      b[i][j]--; //Equivalent to: b[i][j] = b[i][j] – 1
   }
}
```

The following code fragment displays all elements of array `b`.

```
for (i = 0; i <= 9; i++) {
   for (j = 0; j <= 9; j++) {
      System.out.print(b[i][j] + "\t");
   }
   System.out.println();
}
```

✎ *The* `System.out.println()` *statement is used to "display" a line break between rows.*

Second approach

There is also another approach that is very simple but not as flexible as the previous one. There are cases where it cannot be used, as you will see below. Following is a code fragment, written in general form

```
for (type[] row : array_name) {
   for (type element : row) {

      process element;
   }
}
```

in which *process* is any Java statement or block of statements that processes one element of the array at each iteration.

The following Java program, displays all elements of array `b`, one at each iteration.

```
for (String[] row : b) {
   for (String element : row) {
```

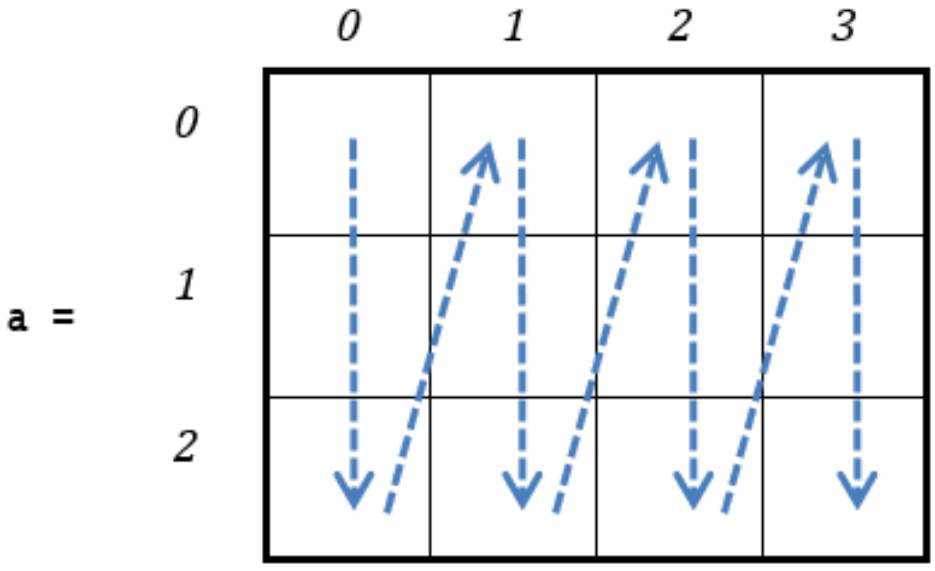
```
      System.out.print(element + "\t");
   }
   System.out.println();
}
```

Keep in mind, though, that this approach cannot be used to alter the values of the elements in an array. For example, if you wish to double the values of all elements in the array `numbers`, you can**not** do the following:

```
int[][] numbers = {
   {5, 10, 3, 2},
   {2, 4, 1, 6}
};
for (int[] x : numbers) {
   for (int number : x) {
      number = number * 2;
   }
}
```

Iterating Through Columns

Iterating through columns means that column 0 is processed first, column 1 is processed next, column 2 afterwards, and so on. Next there is an example of a 3 × 4 array. The arrows show the order in which the elements are processed.



When iterating through columns, the elements of the array are processed as follows:

- the elements of column 0 are processed in the following order

  `a[0][0]` → `a[1][0]` → `a[2][0]`

- the elements of column 1 are processed in the following order

  `a[0][1]` → `a[1][1]` → `a[2][1]`
- the elements of column 2 are processed in the following order

  `a[0][2]` → `a[1][2]` → `a[2][2]`
- the elements of column 3 are processed in the following order

  `a[0][3]` → `a[1][3]` → `a[2][3]`

Using Java statements, let's try to process all elements of a 3 × 4 array (3 rows × 4 columns) by iterating through columns.

```
j = 0; //Variable j refers to column 0.
for (i = 0; i <= 2; i++) { //This loop control structure processes all elements of column 0
    process a[i][j];
}
j = 1; //Variable j refers to column 1.
for (i = 0; i <= 2; i++) { //This loop control structure processes all elements of column 1
    process a[i][j];
}
j = 2; //Variable j refers to column 2.
for (i = 0; i <= 2; i++) { //This loop control structure processes all elements of column 2
    process a[i][j];
}
j = 3; //Variable j refers to column 3.
for (i = 0; i <= 2; i++) { //This loop control structure processes all elements of column 3
    process a[i][j];
}
```

Of course, the same result can be achieved using a nested loop control structure as shown next.

```
for (j = 0; j <= 3; j++) {
   for (i = 0; i <= 2; i++) {
       process a[i][j];
   }
}
```

As you can see, this code fragment differs at only one point from the one that iterates through rows: the two for-loops have switched places. Be careful though. Never switch the places of the two index variables `i` and `j` in the statement *process* `a[i][j]`. Consider the following code fragment, for example. It attempts to iterate through columns in a 3 × 4 array (3 rows × 4 columns) but it does not satisfy the property of definiteness. Can you find out why?

```
for (j = 0; j <= 3; j++) {
   for (i = 0; i <= 2; i++) {

      process a[j][i];

   }
}
```

The trouble arises when variable `j` becomes equal to 3. The statement *process* `a[j][i]` tries to process the elements at **row** index 3 (this is the fourth row) which, of course, does not exist! Still confused? Don't be! There is no row index 3 in a 3 × 4 array! Since row index numbering starts at 0, only rows 0, 1, and 2 actually exist!

32.4 How to Add User-Entered Values to a Two-Dimensional Array

Just as in one-dimensional arrays, instead of reading a value entered from the keyboard and assigning that value to a variable, you can directly assign that value to a specific array element. The following code fragment creates the two-dimensional array `names`, prompts the user to enter six values, and assigns those values to the elements of the array.

```
public static void main(String[] args) throws Exception {
   String[][] names = new String[3][2];
   System.out.print("Name for row 0, column 0: ");
   names[0][0] = cin.nextLine();
   System.out.print("Name for row 0, column 1: ");
   names[0][1] = cin.nextLine();
   System.out.print("Name for row 1, column 0: ");
   names[1][0] = cin.nextLine();
   System.out.print("Name for row 1, column 1: ");
   names[1][1] = cin.nextLine();
   System.out.print("Name for row 2, column 0: ");
   names[2][0] = cin.nextLine();
   System.out.print("Name for row 2, column 1: ");
   names[2][1] = cin.nextLine();
…
```

Using nested for-loops, this code fragment can equivalently be written as

```
static final int ROWS = 3;
static final int COLUMNS = 2;
public static void main(String[] args) throws Exception {
   int i, j;
   String[][] names = new String[ROWS][COLUMNS];
   for (i = 0; i <= ROWS - 1; i++) {
      for (j = 0; j <= COLUMNS - 1; j++) {
         System.out.print("Name for row " + i + ", column " + j + ": ");
         names[i][j] = cin.nextLine();
      }
   }
…
```

Exercise 32.4-1 Displaying Reals Only

Write a Java program that prompts the user to enter numeric values in a 5 × 7 array and then displays the indexes of the elements that contain reals.

Solution

Iterating through rows is the most popular approach, so let's use it. The solution is as follows.

Project_32.4-1

```
static final int ROWS = 5;
static final int COLUMNS = 7;
public static void main(String[] args) throws Exception {
   int i, j;
   double[][] a = new double[ROWS][COLUMNS];
   for (i = 0; i <= ROWS - 1; i++) {
      for (j = 0; j <= COLUMNS - 1; j++) {
         System.out.print("Enter a value for element " + i + ", " + j + ": ");
         a[i][j] = Double.parseDouble(cin.nextLine());
      }
   }
   for (i = 0; i <= ROWS - 1; i++) {
      for (j = 0; j <= COLUMNS - 1; j++) {
         if (a[i][j] != (int)(a[i][j])) { //Check if it is real (float)
            System.out.println("A real found at position: " + i + ", " + j);
         }
      }
   }
}
```

Exercise 32.4-2 Displaying Odd Columns Only

Write a Java program that prompts the user to enter numeric values in a 5 × 7 array and then displays the elements of the columns with odd-numbered indexes (that is, column indexes 1, 3, and 5).

Solution

The Java program is presented next.

Project_32.4-2

```
static final int ROWS = 5;
static final int COLUMNS = 7;
public static void main(String[] args) throws Exception {
   int i, j;
   double[][] a = new double[ROWS][COLUMNS];
   for (i = 0; i <= ROWS - 1; i++) {
      for (j = 0; j <= COLUMNS - 1; j++) {
         System.out.print("Enter a value for element " + i + ", "+ j + ": ");
         a[i][j] = Double.parseDouble(cin.nextLine());
      }
   }
   //Iterate through columns
   for (j = 1; j <= COLUMNS - 1; j += 2) { //Start from 1 and increment by 2
      for (i = 0; i <= ROWS - 1; i++) {
         System.out.print(a[i][j] + " ");
      }
   }
}
```

✎ *This book tries to use, as often as possible, variable* `i` *as the row index and variable* `j` *as the column index. Of course, you can use other variable names as well, such as* `row`, `r` *for row index, or* `column`, `c` *for column index, but variables* `i` *and* `j` *are widely used by the majority of programmers. After using them for a while, your brain will relate* `i` *to rows and* `j` *to columns. Thus, every algorithm or program that uses these variable names as indexes in two-dimensional arrays will be more readily understood.*

32.5 What's the Story on Variables `i` and `j`?

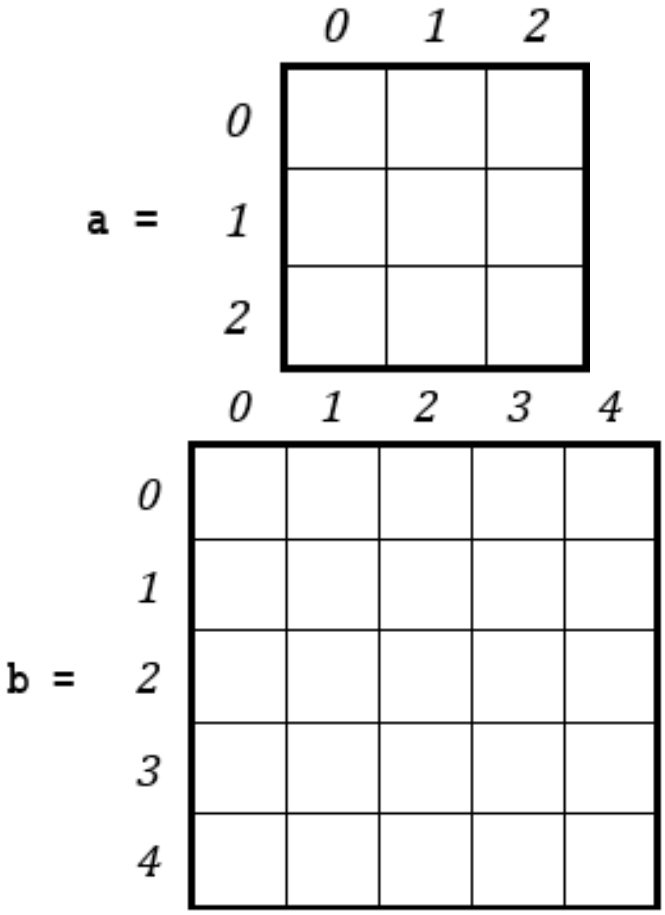
Many programmers believe that the name `i` stands for “index” and `j` is used just because it is after `i`. Others believe that the name `i` stands for “integer”. Probably the truth lies somewhere in the middle.

Mathematicians were using i, j, and k to designate integers in mathematics long before computers were around. Later, in FORTRAN, one of the first

high-level computer languages, variables `i`, `j`, `k`, `l`, `m`, and `n` were integers by default. Thus, the first programmers picked up the habit of using variables `i` and `j` in their programs and it became a convention in most computer languages.

32.6 Square Matrices

In mathematics, a matrix that has the same number of rows and columns is called a *square matrix.* Following are some examples of square matrices.



Exercise 32.6-1 Finding the Sum of the Elements on the Main Diagonal

Write a Java program that lets the user enter numeric values into a 10 × 10 array and then calculates the sum of the elements on its main diagonal.

Solution

In mathematics, the main diagonal of a square matrix is the collection of those elements that runs from the top left corner to the bottom right corner. Following are some examples of square matrices with their main diagonals highlighted by a dark background.

a =

	0	1	2
0	**10**	12	11
1	23	**50**	9
2	12	11	**−3**

b =

	0	1	2	3	4
0	**−3**	44	−2	25	22
1	10	**−1**	29	12	−9
2	5	−3	**16**	22	−8
3	11	25	12	**25**	−5
4	12	22	53	44	**−5**

✎ *Note that the elements on the main diagonal have their row index equal to their column index.*

You can calculate the sum of the elements on the main diagonal using two different approaches. Let's study them both.

First approach – Iterating through all elements

In this approach, the program iterates through rows and checks if the row index is equal to the column index. For square matrices (in this case, arrays) represented as N × N, the number of rows and columns is equal, so you can define just one constant, `N`. The solution is as follows.

Project_32.6-1a

```
static final int N = 10;
public static void main(String[] args) throws Exception {
   int i, j;
   double total;
   double[][] a = new double[N][N];
   for (i = 0; i <= N - 1; i++) {
      for (j = 0; j <= N - 1; j++) {
```

```
            a[i][j] = Double.parseDouble(cin.nextLine());
         }
      }
      //Calculate the sum
      total = 0;
      for (i = 0; i <= N - 1; i++) {
         for (j = 0; j <= N - 1; j++) {
            if (i == j) {
               total += a[i][j]; //This is equivalent to: total = total + a[i][j]
            }
         }
      }
      System.out.println("Sum = " + total);
}
```

✎ *Note that the program iterates through rows and checks if the row index is equal to the column index. Alternatively, the same result can be achieved by iterating through columns.*

✎ *In this approach, the nested loop control structure that is responsible for calculating the sum performs 10 × 10 = 100 iterations.*

Second approach – Iterating directly through the main diagonal

In this approach, one single loop control structure iterates directly through the main diagonal. The solution is as follows.

Project_32.6-1b

```
static final int N = 10;
public static void main(String[] args) throws Exception {
   int i, j, k;
   double total;
   double[][] a = new double[N][N];
   for (i = 0; i <= N - 1; i++) {
      for (j = 0; j <= N - 1; j++) {
         a[i][j] = Double.parseDouble(cin.nextLine());
      }
   }
   //Calculate the sum
   total = 0;
   for (k = 0; k <= N - 1; k++) {
      total += a[k][k];
   }
   System.out.println("Sum = " + total);
}
```

✎ *This approach is much more efficient than the first one since the total number of iterations performed by the for-loop that is responsible for calculating the sum is just 10.*

Exercise 32.6-2 Finding the Sum of the Elements on the Antidiagonal

Write a Java program that lets the user enter numeric values in a 5 × 5 array and then calculates the sum of the elements on its antidiagonal.

Solution

In mathematics, the antidiagonal of a square matrix is the collection of those elements that runs from the top right corner to the bottom left corner of the array. Next, you can find an example of a 5 × 5 square matrix with its antidiagonal highlighted by a dark background.

a =

	0	1	2	3	4
0	−3	44	−2	25	**22**
1	10	−1	29	**12**	−9
2	5	−3	**16**	22	−8
3	11	**25**	12	25	−5
4	**12**	22	53	44	−5

The indexes of any element on the antidiagonal of an N × N array satisfy the following equation:

```
i + j = N - 1
```

where variables `i` and `j` correspond to the row and column indexes respectively.

If you solve for `j`, the equation becomes:

```
j = N - i - 1
```

Using this formula, you can calculate the indexes of any element on the antidiagonal; that is, for any value of variable `i`, you can find the corresponding value of variable `j`. For example, in the previous 5 × 5 square array where N equals 5, when `i` is 3 the value of variable `j` is:

$$j = N - i - 1 \iff j = 5 - 3 - 1 \iff j = 1$$

Using all this knowledge, let's now write the corresponding Java program.

Project_32.6-2

```
static final int N = 5;
public static void main(String[] args) throws Exception {
   int i, j;
   double total;
   double[][] a = new double[N][N];
   for (i = 0; i <= N - 1; i++) {
      for (j = 0; j <= N - 1; j++) {
         a[i][j] = Double.parseDouble(cin.nextLine());
      }
   }
   //Calculate the sum
   total = 0;
   for (i = 0; i <= N - 1; i++) {
      j = N - i - 1; //Equivallent to:
      total += a[i][j]; //total += a[i][N - i - 1];
   }
   System.out.println("Sum = " + total);
}
```

✎ *Note that the for-loop that is responsible for finding the sum of the elements on the antidiagonal iterates directly through the antidiagonal.*

Exercise 32.6-3 Filling in the Array

Write a Java program that creates and displays the following array.

a =

	0	*1*	*2*	*3*	*4*
0	−1	20	20	20	20
1	10	−1	20	20	20
2	10	10	−1	20	20
3	10	10	10	−1	20
4	10	10	10	10	−1

Solution

As you can see, there is the value of −1 in the main diagonal. You already know that the common characteristic between the elements of the main diagonal of a square matrix is that they have their row index equal to their column index. Now, what you also need is to find a common characteristic between all elements that contain the value 10, and another such common characteristic between all elements that contain the value 20. And actually there are! The row index of any element containing the value 10 is, in every case, greater than its corresponding column index, and similarly, the row index of any element containing the value 20 is, in every case, less than its corresponding column index.

Accordingly, the Java program is as follows.

Project_32.6-3

```
static final int N = 5;
public static void main(String[] args) throws Exception {
   int i, j;
   int[][] a = new int[N][N];
   for (i = 0; i <= N - 1; i++) {
      for (j = 0; j <= N - 1; j++) {
         if (i == j) {
            a[i][j] = -1;
         }
         else if (i > j) {
            a[i][j] = 10;
         }
         else {
            a[i][j] = 20;
         }
      }
   }
   for (i = 0; i <= N - 1; i++) {
      for (j = 0; j <= N - 1; j++) {
         System.out.print(a[i][j] + "\t");
      }
      System.out.println();
   }
}
```

32.7 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) All the elements of a two-dimensional array must contain different values.

2) In order to refer to an element of a two-dimensional array you need two indexes.
3) The two indexes of a two-dimensional array must be either both variables, or both constant values.
4) A 5 × 6 array is a two-dimensional array that has five columns and six rows.
5) To refer to an element of array `b` that exists at the second row and third column, you would write `b[2][3]`.
6) Iterating through rows means that first row of a two-dimensional array is processed first, the second row is process next, and so on.
7) You cannot use variables other than `i` and `j` to iterate through a two-dimensional array.
8) The following Java statement creates a two-dimensional array.

```
int[][] names = new int[3][7];
```

9) The following code fragment creates a two-dimensional array of four elements and assigns values to them.

```
String[][] names = new String[2][2];
names[0][0] = "John";
names[0][1] = "George";
names[1][0] = "Sally";
names[1][1] = "Angelina";
```

10) The following code fragment assigns the value 10 to an element that exists in the row with index 0.

```
values[0][0] = 7;
values[0][values[0][0]] = 10;
```

11) The following statement adds the name “Sally” to an element that exists in the row with index 1.

```
String[][] names = {{"John", "George"}, {"Sally", "Angelina"}};
```

12) The following code fragment displays the name “Sally” on the screen.

```
String[][] names = new String[2][2];
k = 0;
names[0][k] = "John";
k++;
names[0][k] = "George";
names[1][k] = "Sally";
k--;
names[1][k] = "Angelina";
```

```
System.out.println(names[1][1]);
```

13) The following code fragment satisfies the property of definiteness.

```
String[][] grades = {{"B+", "A+"}, {"A", "C-"}};
System.out.println(grades[2][2]);
```

14) The following code fragment satisfies the property of definiteness.

```
int[][] values = {{1, 0}, {2, 0}};
System.out.println(values[values[0][0]][values[0][1]]);
```

15) The following code fragment displays the value 2 on the screen.

```
int[][] values = {{0, 1}, {2, 0}};
System.out.println(values[values[0][1]][values[0][0]]);
```

16) The following code fragment displays all the elements of a 3 × 4 array.

```
for (k = 0; k <= 11; k++) {
   i = (int)(k / 4);
   j = k % 4;
   System.out.println(names[i][j]);
}
```

17) The following code fragment lets the user enter 100 values into array `b`.

```
for (i = 0; i <= 9; i++) {
   for (j = 0; j <= 9; j++) {
      b[i][j] = cin.nextLine();
   }
}
```

18) If array `b` contains 10 × 20 elements, the following code fragment doubles the values of all of its elements.

```
for (i = 9; i >= 0; i--) {
   for (j = 19; j >= 0; j--) {
      b[i][j] *= 2;
}
```

19) If array `b` contains 10 × 20 elements, the following code fragment displays some of them.

```
for (i = 0; i <= 8; i += 2) {
   for (j = 0; j <= 19; j++) {
      System.out.println(b[i][j]);
   }
}
for (i = 1; i <= 9; i += 2) {
   for (j = 0; j <= 19; j++) {
      System.out.println(b[i][j]);
   }
}
```

20) The following code fragment displays only the columns with even-numbered indexes.

```
for (j = 0; j <= 10; j += 2) {
   for (i = 0; i <= 9; i++) {
      System.out.println(a[i][j]);
   }
}
```

21) A 5 × 5 array is a square array.

22) In the main diagonal of a N × N array, all elements have their row index equal to their column index.

23) In mathematics, the antidiagonal of a square matrix is the collection of those elements that runs from the top left corner to the bottom right corner of the array.

24) Any element on the antidiagonal of an N × N array satisfies the equation `i + j = N − 1`, where variables `i` and `j` correspond to the row and column indexes respectively.

25) The following code fragment calculates the sum of the elements on the main diagonal of a N × N array.

```
total = 0;
for (k = 0; k <= N - 1; k++) {
   total += a[k][k];
}
```

26) The following code fragment displays all the elements of the antidiagonal of an N × N array.

```
for (i = N - 1; i >= 0; i--) {
   System.out.println(a[i][N - i - 1]);
}
```

27) The column index of any element of a N × N array that is below the main diagonal is always greater than its corresponding row index.

32.8 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The following statement

```
String lastNames = String[5][4];
```

   a) contains logic error(s).

   b) contains syntax error(s).

c) is a correct statement.

d) none of the above

2) The following code fragment

```
int[][] values = {{1, 0} {2, 0}};
System.out.println(values[values[0][0], values[0][1]]);
```

a) contains logic error(s).

b) contains syntax error(s).

c) none of the above

3) The following code fragment

```
x = Integer.parseInt(cin.nextLine());
y = Integer.parseInt(cin.nextLine());
names[x][y] = 10;
```

a) does not satisfy the property of finiteness.

b) does not satisfy the property of effectiveness.

c) does not satisfy the property of definiteness.

d) none of the above

4) If variable `x` contains the value 4, the following statement

```
names[x + 1][x] = 5;
```

a) assigns the value 5 to the element with row index 5 and column index 4.

b) assigns the value 5 to the element with row index 4 and column index 5.

c) assigns the value 5 to the element with row index 5 and column index 5.

d) none of the above

5) The following statement

```
int[][] names = {{3, 5, 2}};
```

a) assigns the value 5 to the element with row index 0 and column index 1.

b) assigns the value 3 to the element with row index 0 and column index 0.

c) assigns the value 2 to the element with row index 0 and column index 2.

d) all of the above

e) none of the above

6) The following statement

```
int[][] values = new int[1][2];
```

a) creates a 1 × 2 array.

b) creates a 2 × 1 array.

c) creates a one-dimensional array.

d) none of the above

7) You can iterate through a two-dimensional array with two nested loop control structures that use

a) variables `i` and `j` as counters.

b) variables `k` and `l` as counters.

c) variables `m` and `n` as counters.

d) any variables as counters.

8) The following code fragment

```
String[][] names = {{"John", "Sally"}, {"George", "Maria"}};
for (j = 0; j <= 1; j++) {
   for (i = 1; i >= 0; i--) {
      System.out.println(names[i][j]);
   }
}
```

a) displays all names in descending order.

b) displays some names in descending order.

c) displays all names in ascending order.

d) displays some names in ascending order.

e) none of the above

9) If array `b` contains 30 × 40 elements, the following code fragment

```
for (i = 30; i >= 1; i--) {
   for (j = 40; j >= 1; j--) {
      b[i][j] *= 3;
   }
}
```

a) triples the values of some of its elements.

b) triples the values of all of its elements.

c) does not satisfy the property of definiteness.

d) none of the above

10) If array `b` contains 30 × 40 elements, the following code fragment

```
total = 0;
for (i = 29; i >= 0; i--) {
   for (j = 39; j >= 0; j--) {
      total += b[i][j];
   }
}
average = total / 120;
```

a) calculates the sum of all of its elements.

b) calculates the average value of all of its elements.

c) all of the above

11) The following two code fragments calculate the sum of the elements on the main diagonal of an N × N array,

```
total = 0;
for (i = 0; i <= N - 1; i++) {
   for (j = 0; j <= N - 1; j++) {
      if (i == j) {
         total += a[i][j];
      }
   }
}
total = 0;
for (k = 0; k <= N - 1; k++) {
   total += a[k][k];
}
```

a) but the first one is more efficient.

b) but the second one is more efficient.

c) none of the above; both code fragments perform equivalently

32.9 Review Exercises

Complete the following exercises.

1) Create the trace table for the following code fragment.

```
int[][] a = new int[2][3];
a[0][2] = 1;
x = 0;
a[0][x] = 9;
a[0][x + a[0][2]] = 4;
```

```
a[a[0][2]][2] = 19;
a[a[0][2]][x + 1] = 13;
a[a[0][2]][x] = 15;
```

2) Create the trace table for the following code fragment.

```
int[][] a = new int[2][3];
for (i = 0; i <= 1; i++) {
   for (j = 0; j <= 2; j++) {
      a[i][j] = (i + 1) * 5 + j;
   }
}
```

3) Create the trace table for the following code fragment.

```
int[][] a = new int[3][3];
for (j = 0; j <= 2; j++) {
   for (i = 0; i <= 2; i++) {
      a[i][j] = (i + 1) * 2 + j * 4;
   }
}
```

4) Try, without using a trace table, to determine the values that the array will contain when the following code fragment is executed. Do this for three different executions. The corresponding input values are: (i) 5, (ii) 9, and (iii) 3.

```
int[][] a = new int[2][3];
x = Integer.parseInt(cin.nextLine());
for (i = 0; i <= 1; i++) {
   for (j = 0; j <= 2; j++) {
      a[i][j] = (x + i) * j;
   }
}
```

5) Try, without using a trace table, to determine the values that the array will contain when the following code fragment is executed. Do this for three different executions. The corresponding input values are: (i) 13, (ii) 10, and (iii) 8.

```
int[][] a = new int[2][3];
x = Integer.parseInt(cin.nextLine());
for (i = 0; i <= 1; i++) {
   for (j = 0; j <= 2; j++) {
      if (j < x % 4)
         a[i][j] = (x + i) * j;
      else
         a[i][j] = (x + j) * i + 3;
   }
```

```
}
```

6) Try, without using a trace table, to determine the values that the array will contain when the following code fragment is executed.

```
double[][] a = {{18, 10, 35}, {32, 12, 19}};
for (j = 0; j <= 2; j++) {
   for (i = 0; i <= 1; i++) {
      if (a[i][j] < 13)
         a[i][j] /= 2;
      else if (a[i][j] < 20)
         a[i][j]++;
      else
         a[i][j] -= 4;
   }
}
```

7) Try, without using a trace table, to determine the values that the array will contain when the following code fragment is executed.

```
int[][] a = {{11, 10}, {15, 19}, {22, 15}};
for (j = 0; j <= 1; j++) {
   for (i = 0; i <= 2; i++) {
      if (i == 2)
         a[i][j] += a[i - 1][j];
      else
         a[i][j] += a[i + 1][j];
   }
}
```

8) Assume that array a contains the following values.

a =

	0	1	2	3
0	−1	15	22	3
1	25	12	16	14
2	7	9	1	45
3	40	17	11	13

What displays on the screen after executing each of the following code fragments?

i)

```
for (i = 0; i <= 2; i++) {
   for (j = 0; j <= 2; j++) {
```

```
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

ii)

```
for (i = 2; i >= 0; i--) {
   for (j = 0; j <= 2; j++) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

iii)

```
for (i = 0; i <= 2; i++) {
   for (j = 2; j >= 0; j--) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

iv)

```
for (i = 2; i >= 0; i--) {
   for (j = 2; j >= 0; j--) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

v)

```
for (j = 0; j <= 2; j++) {
   for (i = 0; i <= 2; i++) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

vi)

```
for (j = 0; j <= 2; j++) {
   for (i = 2; i >= 0; i--) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

vii)

```
for (j = 2; j >= 0; j--) {
```

```
   for (i = 0; i <= 2; i++) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

viii)

```
for (j = 2; j >= 0; j--) {
   for (i = 2; i >= 0; i--) {
      System.out.print(a[i][j]);
      System.out.print(" ");
   }
}
```

9) Write a Java program that lets the user enter integer values in a 10 × 15 array and then displays the indexes of the elements that contain odd numbers.

10) Write a Java program that lets the user enter numeric values in a 10 × 6 array and then displays the elements of the columns with even-numbered indexes (that is, column indexes 0, 2, and 4).

11) Write a Java program that lets the user enter numeric values in a 12 × 8 array and then calculates and displays the sum of the elements that have even column indexes and odd row indexes.

12) Write a Java program that lets the user enter numeric values in an 8 × 8 square array and then calculates the average value of the elements on its main diagonal and the average value of the elements on its antidiagonal. Try to calculate both average values within the same loop control structure.

13) Write a Java program that creates and displays the following array.

a =

	0	1	2	3	4
0	11	11	11	11	5
1	11	11	11	5	88
2	11	11	5	88	88
3	11	5	88	88	88
4	5	88	88	88	88

14) Write a Java program that creates and displays the following array.

a =

	0	1	2	3	4
0	0	11	11	11	5
1	11	0	11	5	88
2	11	11	0	88	88
3	11	5	88	0	88
4	5	88	88	88	0

15) Write a Java program that lets the user enter numeric values in a 5 × 4 array and then displays the row and column indexes of the elements that contain integers.

16) Write a Java program that lets the user enter numeric values in a 10 × 4 array and then counts and displays the total number of negative elements.

17) Write a Java program that lets the user enter words in a 3 × 4 array and then displays them with a space character between them.

18) Write a Java program that lets the user enter words in a 20 × 14 array and then displays those who have less than five characters.

    Hint: Use the `length()` method.

19) Write a Java program that lets the user enter words in a 20 × 14 array and displays those that have less than 5 characters, then those that have less than 10 characters, and finally those that have less than 20 characters.

Hint: Try to display the words using three for-loops nested one within the other.

Chapter 33
Tips and Tricks with Data Structures

33.1 Introduction

Since arrays are handled with the same sequence, decision, and loop control structures that you learned about in previous chapters, there is no need to repeat all of that information here. However, what you will discover in this chapter is how to process each row or column of a two-dimensional array individually, how to solve problems that require the use of more than one array, how to create a two-dimensional array from a one-dimensional array (and vice versa), and some useful built-in array methods that Java supports.

33.2 Processing Each Row Individually

Processing each row individually means that every row is processed separately and the result of each row (which can be the sum, the average value, and so on) can be used individually for further processing.

Suppose you have the following 4 × 5 array.

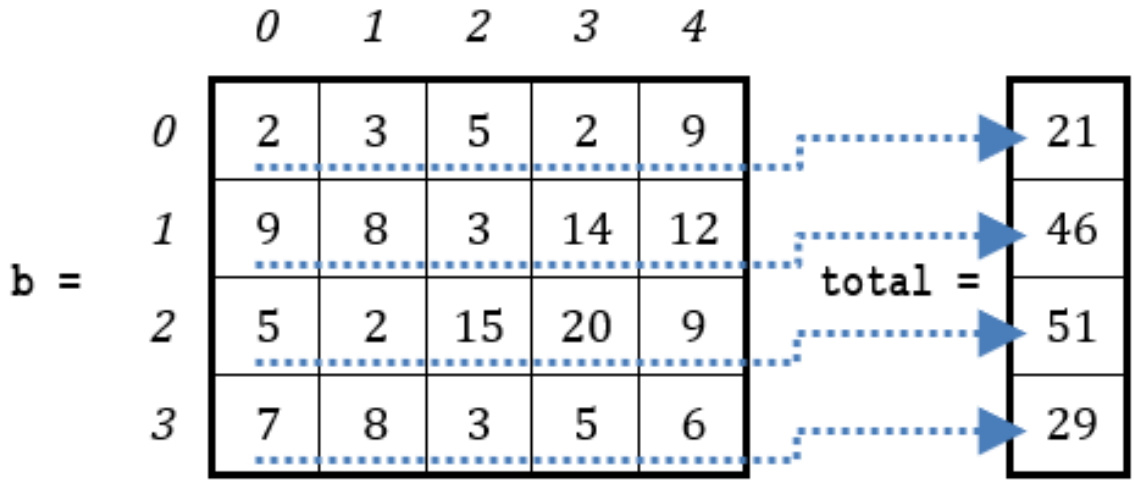
b =

	0	1	2	3	4
0	2	3	5	2	9
1	9	8	3	14	12
2	5	2	15	20	9
3	7	8	3	5	6

Let's try to find the sum of each row individually. Both of the following approaches iterate through rows.

First approach – Creating an auxiliary array

In this approach, the program processes each row individually and creates an auxiliary array in which each element stores the sum of one row. This approach gives you much flexibility since you can use this new array later in your program for further processing. The auxiliary array `total` is shown on the right.



Now, let's write the corresponding code fragment. To more easily understand the process, the “from inner to outer” method is used. The following code fragment calculates the sum of the first row (row index 0) and stores the result in the element at position 0 of the auxiliary array `total`. Assume variable `i` contains the value 0.

```
s = 0;
for (j = 0; j <= COLUMNS - 1; j++) {
   s += b[i][j];
}
total[i] = s;
```

This code fragment can equivalently be written as

```
total[i] = 0;
for (j = 0; j <= COLUMNS - 1; j++) {
   total[i] += b[i][j];
}
```

Now, nesting this code fragment in a for-loop that iterates for all rows results in the following.

```
for (i = 0; i <= ROWS - 1; i++) {
   total[i] = 0;
   for (j = 0; j <= COLUMNS - 1; j++) {
      total[i] += b[i][j];
   }
}
```

Second approach – Just find it and process it.

This approach uses no auxiliary array; it just calculates and directly processes the sum. The code fragment is as follows.

```
for (i = 0; i <= ROWS - 1; i++) {
   total = 0;
```

```
   for (j = 0; j <= COLUMNS - 1; j++) {
      total += b[i][j];
   }

   process total;
}
```

What does *process* `total` mean? It depends on the given problem. It may just display the sum, it may calculate the average value of each individual row and display it, or it may use the sum for calculating even more complex mathematical expressions.

For instance, the following example calculates and displays the average value of each row of array `b`.

```
for (i = 0; i <= ROWS - 1; i++) {
   total = 0;
   for (j = 0; j <= COLUMNS - 1; j++) {
      total += b[i][j];
   }
   average = total / COLUMNS;
   System.out.println(average);
}
```

Exercise 33.2-1 Finding the Average Value

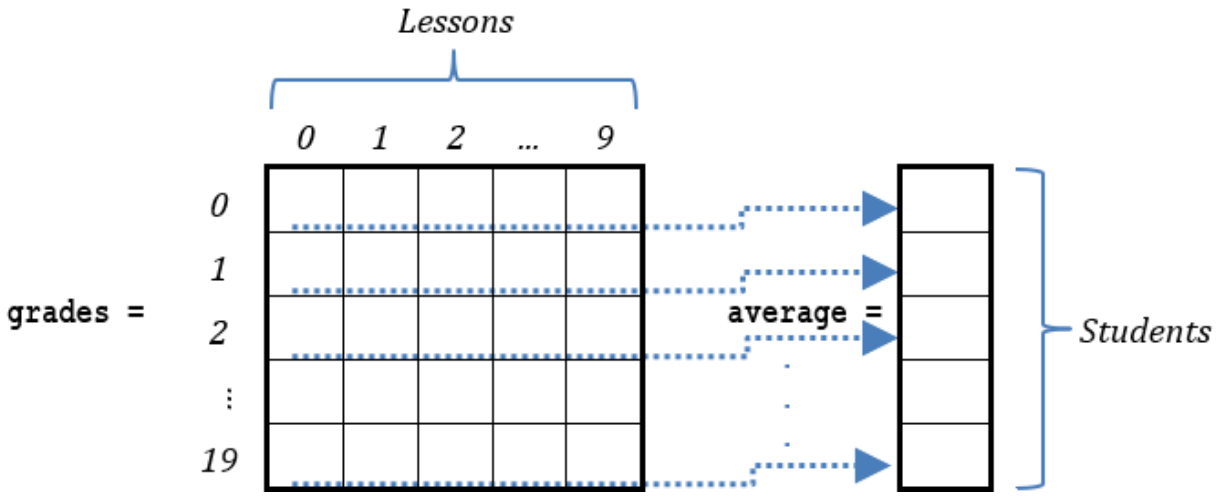
There are 20 students and each one of them has received their grades for 10 lessons. Write a Java program that prompts the user to enter the grades of each student for all lessons and then calculates and displays, for each student, all average values that are greater than 89.

Solution

Since you've learned two approaches for processing each row individually, let's use them both.

First approach – Creating an auxiliary array

In this approach, the program processes each row individually and creates an auxiliary array in which each element stores the average value of one row. The two required arrays are shown next.



After the array average is created, the program can find and display all average values that are greater than 89. The Java program is as follows.

Project_33.2-1a

```
static final int STUDENTS = 20;
static final int LESSONS = 10;
public static void main(String[] args) throws Exception {
   int i, j;
   int[][] grades = new int[STUDENTS][LESSONS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.println("For student No. " + (i + 1) + "...");
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("enter grade for lesson No. " + (j + 1) + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Create array average. Iterate through rows
   double[] average = new double[STUDENTS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      average[i] = 0;
      for (j = 0; j <= LESSONS - 1; j++) {
         average[i] += grades[i][j];
      }
      average[i] /= LESSONS;
   }
   //Display all average values that are greater than 89
   for (i = 0; i <= STUDENTS - 1; i++) {
      if (average[i] > 89) {
         System.out.println(average[i]);
      }
   }
}
```

Second approach – Just find it and display it!

This approach uses no auxiliary array; it just calculates and directly displays all average values that are greater than 89. The Java program is as follows.

Project_33.2-1b

```
static final int STUDENTS = 20;
static final int LESSONS = 10;
public static void main(String[] args) throws Exception {
   int i, j;
   double average;
   int[][] grades = new int[STUDENTS][LESSONS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.println("For student No. " + (i + 1) + "...");
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("enter grade for lesson No. " + (j + 1) + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Calculate the average value of each row and directly display those who are greater
   than 89
   for (i = 0; i <= STUDENTS - 1; i++) {
      average = 0;
      for (j = 0; j <= LESSONS - 1; j++) {
         average += grades[i][j];
      }
      average /= LESSONS;
      if (average > 89) {
         System.out.println(average);
      }
   }
}
```

33.3 Processing Each Column Individually

Processing each column individually means that every column is processed separately and the result of each column (which can be the sum, the average value, and so on) can be used individually for further processing. Suppose you have the following 4 × 5 array.

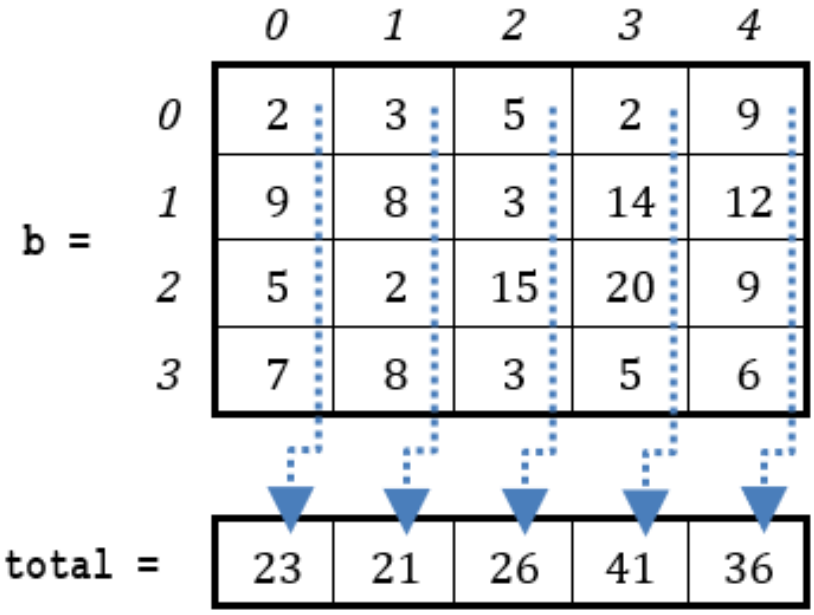
b =

	0	1	2	3	4
0	2	3	5	2	9
1	9	8	3	14	12
2	5	2	15	20	9
3	7	8	3	5	6

As before, let's try to find the sum of each column individually. Yet again, there are two approaches that you can use. Both of these approaches iterate through columns.

First approach – Creating an auxiliary array

In this approach, the program processes each column individually and creates an auxiliary array in which each element stores the sum of one column. This approach gives you much flexibility since you can use this new array later in your program for further processing. The auxiliary array `total` is shown at the bottom.



Now, let's write the corresponding code fragment. To more easily understand the process, the “from inner to outer” method is used again. The following code fragment calculates the sum of the first column (column index 0) and stores the result in the element at position 0 of the auxiliary array `total`. Assume variable `j` contains the value 0.

```
s = 0;
for (i = 0; i <= ROWS - 1; i++) {
   s += b[i][j];
}
total[j] = s;
```

This program can equivalently be written as

```
total[j] = 0;
for (i = 0; i <= ROWS - 1; i++) {
   total[j] += b[i][j];
}
```

Now, nesting this code fragment in a for-loop that iterates for all columns results in the following.

```
for (j = 0; j <= COLUMNS - 1; j++) {
   total[j] = 0;
   for (i = 0; i <= ROWS - 1; i++) {
      total[j] += b[i][j];
   }
}
```

Second approach – Just find it and process it.

This approach uses no auxiliary array; it just calculates and directly processes the sum. The code fragment is as follows.

```
for (j = 0; j <= COLUMNS - 1; j++) {
   total = 0;
   for (i = 0; i <= ROWS - 1; i++) {
      total += b[i][j];
   }
   process total;
}
```

Accordingly, the following code fragment calculates and displays the average value of each column.

```
for (j = 0; j <= COLUMNS - 1; j++) {
   total = 0;
   for (i = 0; i <= ROWS - 1; i++) {
      total += b[i][j];
   }
   System.out.println(total / ROWS);
}
```

Exercise 33.3-1 Finding the Average Value

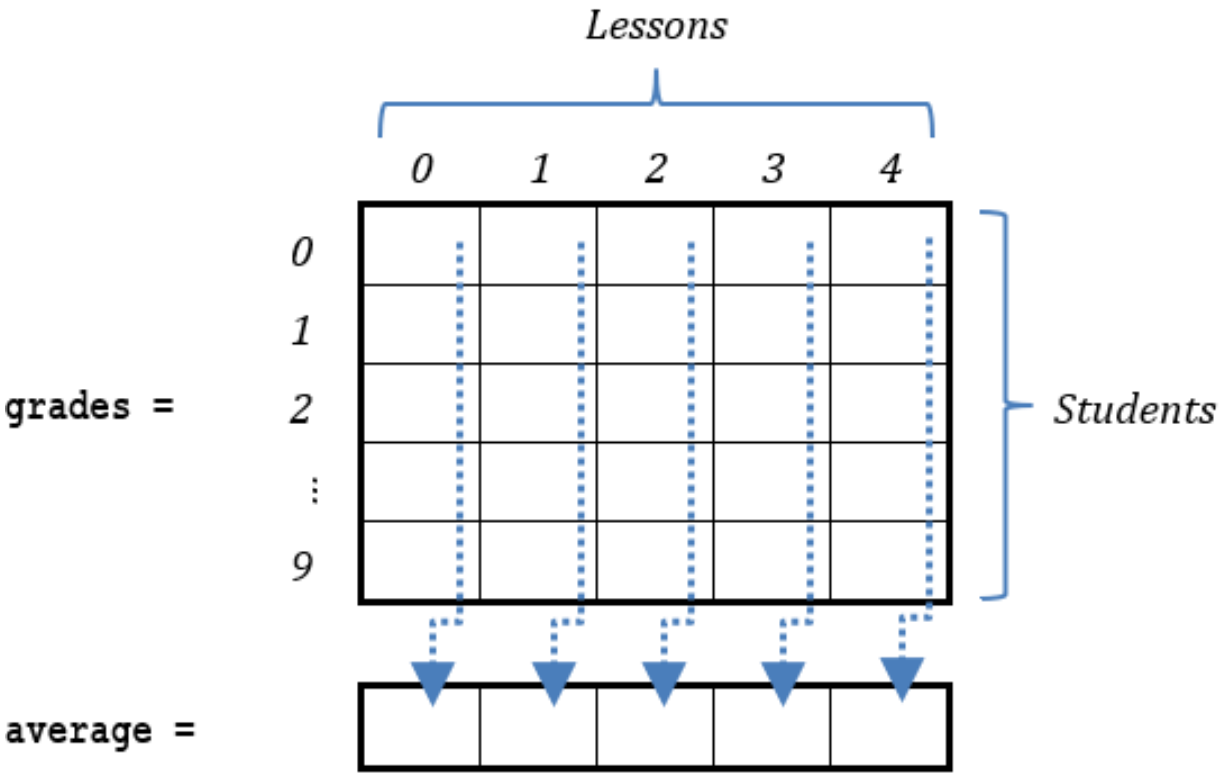
There are 10 students and each one of them has received their grades for five lessons. Write a Java program that prompts the user to enter the grades of each student for all lessons and then calculates and displays, for each lesson, all average values that are greater than 89.

Solution

Since you've learned two approaches for processing each column individually, let's use them both.

First approach – Creating an auxiliary array

In this approach, the program processes each column individually and creates an auxiliary array in which each element stores the average value of one column. The two required arrays are shown next.



After the array `average` is created, the program can find and display all average values that are greater than 89. The Java program is as follows.

Project_33.3-1a

```
static final int STUDENTS = 10;
static final int LESSONS = 5;
public static void main(String[] args) throws Exception {
   int i, j;
   int[][] grades = new int[STUDENTS][LESSONS];
```

```
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.println("For student No. " + (i + 1) + "...");
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("enter grade for lesson No. " + (j + 1) + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Create array average. Iterate through columns
   double[] average = new double[LESSONS];
   for (j = 0; j <= LESSONS - 1; j++) {
      average[j] = 0;
      for (i = 0; i <= STUDENTS - 1; i++) {
         average[j] += grades[i][j];
      }
      average[j] /= STUDENTS;
   }
   //Display all average values than are greater than 89
   for (j = 0; j <= LESSONS - 1; j++) {
      if (average[j] > 89) {
         System.out.println(average[j]);
      }
   }
}
```

Second approach – Just find it and display it!

This approach uses no auxiliary array; it just calculates and directly displays all average values that are greater than 89. The Java program is as follows.

Project_33.3-1b

```
static final int STUDENTS = 10;
static final int LESSONS = 5;
public static void main(String[] args) throws Exception {
   int i, j;
   double average;
   int[][] grades = new int[STUDENTS][LESSONS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.println("For student No. " + (i + 1) + "...");
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("enter grade for lesson No. " + (j + 1) + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Calculate the average value of each column
   //and directly display those who are greater than 89
   for (j = 0; j <= LESSONS - 1; j++) {
      average = 0;
```

```
      for (i = 0; i <= STUDENTS - 1; i++) {
         average += grades[i][j];
      }
      average /= STUDENTS;
      if (average > 89) {
         System.out.println(average);
      }
   }
}
```

33.4 How to Use More Than One Data Structures in a Program

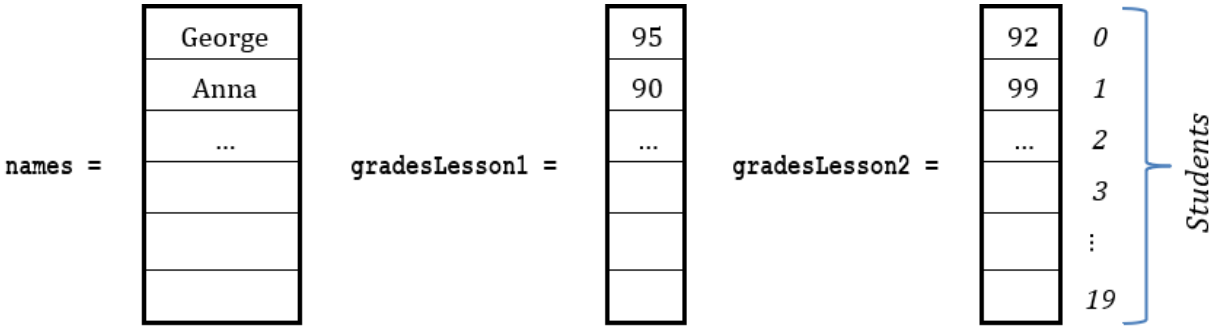
So far, every example or exercise has used just one array or one hashmap. But what if a problem requires you to use two arrays, or one array and one hashmap, or one array and two hashmaps? Next you will find some exercises that show you how various data structures can be combined to tackle a variety of unique challenges.

Exercise 33.4-1 Using Three One-Dimensional Arrays

There are 20 students and each one of them has received grades for two lessons. Write a Java program that prompts the user to enter the name and grades of each student for both lessons. The program must then find and display the names of all students who have grades greater than 89 for both lessons.

Solution

Following are the required arrays containing some typical values.



As you can see, there is a one-to-one correspondence between the elements in the array `names` and those in the arrays `gradesLesson1`, and `gradesLesson2`. The first of the twenty students is George, and he received grades of 95 and 92 for the two lessons. The name “George” is stored at index 0 of the array `names`, and at exactly the same index in the arrays `gradesLesson1` and `gradesLesson2`, his grades for the two lessons are stored.

The next student (Anna) and her grades are stored at index 1 of the arrays `names`, `gradesLesson1`, and `gradesLesson2`, respectively, and so on.

The Java program is as follows.

Project_33.4-1

```
static final int STUDENTS = 20;
public static void main(String[] args) throws Exception {
   int i;
   String[] names = new String[STUDENTS];
   int[] gradesLesson1 = new int[STUDENTS];
   int[] gradesLesson2 = new int[STUDENTS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.print("Enter student name No" + (i + 1) + ": ");
      names[i] = cin.nextLine();
      System.out.print("Enter grade for lesson 1: ");
      gradesLesson1[i] = Integer.parseInt(cin.nextLine());
      System.out.print("Enter grade for lesson 2: ");
      gradesLesson2[i] = Integer.parseInt(cin.nextLine());
   }
   //Display the names of those who have grades greater than 89 for both lessons
   for (i = 0; i <= STUDENTS - 1; i++) {
      if (gradesLesson1[i] > 89 && gradesLesson2[i] > 89) {
         System.out.println(names[i]);
      }
   }
}
```

Exercise 33.4-2 Using a One-Dimensional Array Along with a Two-Dimensional Array

There are 10 students and each one of them has received their grades for five lessons. Write a Java program that prompts the user to enter the name of each student and the grades for all lessons and then calculates and displays the names of the students who have more than one grade greater than 89.

Solution

In this exercise, you could do what you did in the previous one. You could, for example, use a one-dimensional array to store the names of the students and five one-dimensional arrays to store the grades for each student for each lesson. Not very convenient, but it could work. Obviously, when there are more than two grades, this is not the most suitable approach.

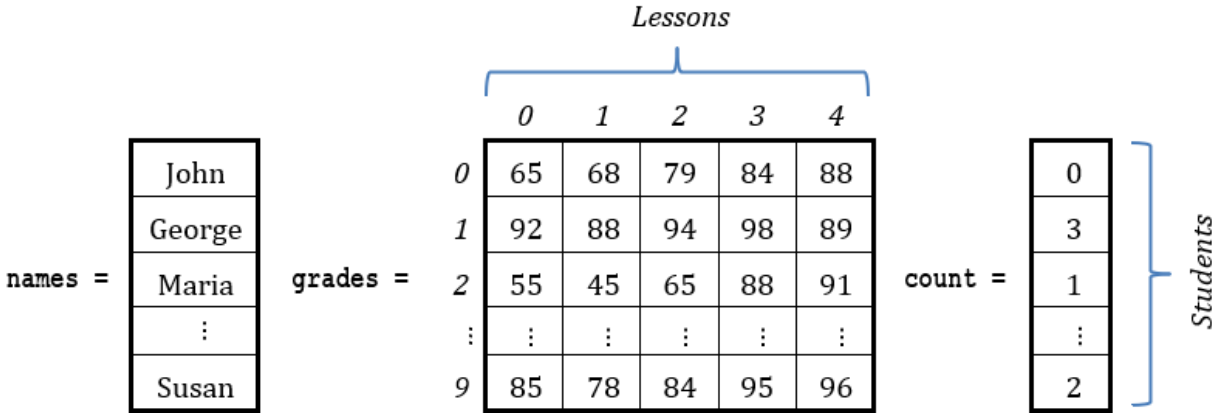
The best approach here is to use a one-dimensional array to store the names of the students and a two-dimensional array to store the grades for each

student for each lesson.

There are actually two approaches. Which one to use depends clearly on you! If you decide that, in the two-dimensional array, the rows should refer to students and the columns should refer to lessons then you can use the first approach discussed below. If you decide that the rows should refer to lessons and the columns should refer to students then you can use the second approach that follows.

First approach – Rows for students, columns for lessons

In this approach, the two-dimensional array must have 10 rows, one for every student and 5 columns, one for every lesson. All other arrays can be placed in relation to this two-dimensional array as follows.



✎ *The auxiliary array* `count` *will be created by the program and will store the number of grades for each student that are greater than 89.*

Now, let's see how to read values and store them in the arrays `names` and `grades`. One simple solution would be to use one for-loop for reading names, and then nested for-loops for reading grades. However, it may not be very practical for the user to first enter all names and then all grades. A more user-friendly approach would be to prompt the user to enter one student name and then all of their grades, then proceed to the next student name and their corresponding grades, and so on. The solution is as follows.

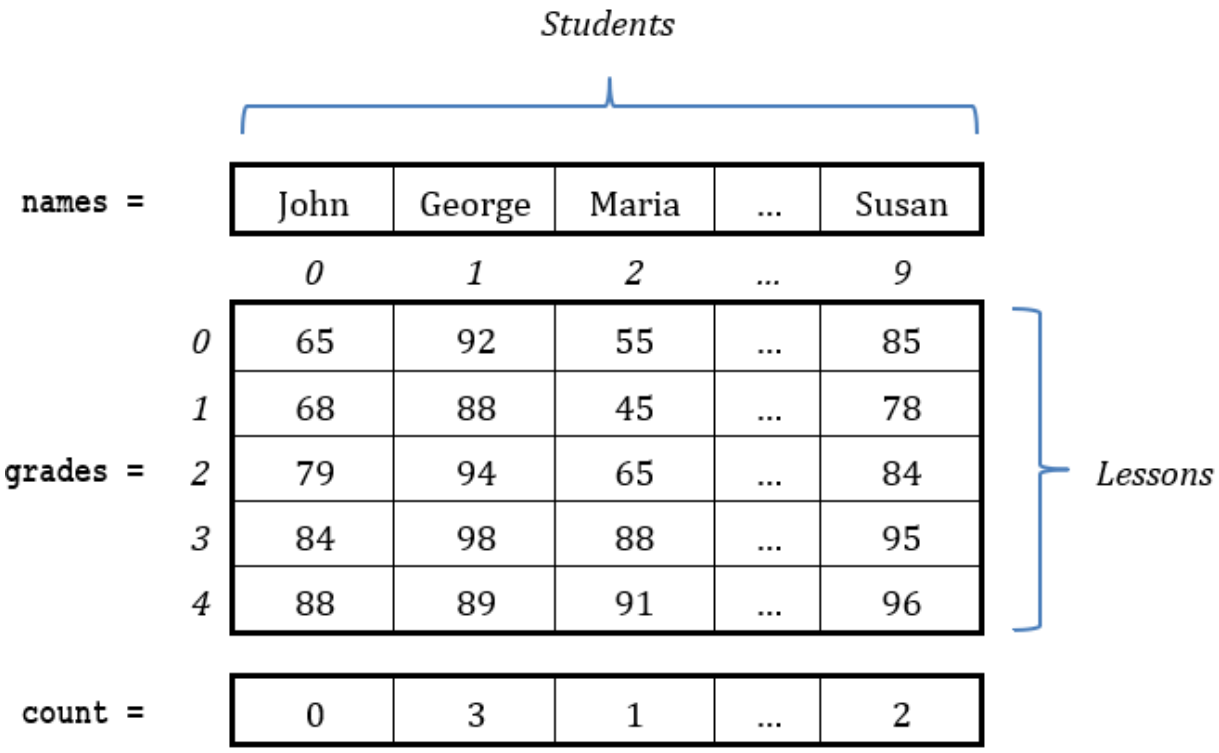
Project_33.4-2a

```
static final int STUDENTS = 10;
static final int LESSONS = 5;
public static void main(String[] args) throws Exception {
   int i, j;
   //Read names and grades all together. Iterate through rows in array grades
   String[] names = new String[STUDENTS];
```

```
   int[][] grades = new int[STUDENTS][LESSONS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.print("Enter name for student No. " + (i + 1) + ": ");
      names[i] = cin.nextLine();
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("Enter grade No. " + (j + 1) + " for " + names[i] + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Create array count. Iterate through rows
   int[] count = new int[STUDENTS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      count[i] = 0;
      for (j = 0; j <= LESSONS - 1; j++) {
         if (grades[i][j] > 89) {
            count[i]++;
         }
      }
   }
   //Displays the names of the students who have more than one grade greater than 89
   for (i = 0; i <= STUDENTS - 1; i++) {
      if (count[i] > 1) {
         System.out.println(names[i]);
      }
   }
}
```

Second approach – Rows for lessons, columns for students

In this approach, the two dimensional array must have 5 rows, one for every lesson and 10 columns, one for every student. All other arrays can be placed in relation to this two-dimensional array, as shown next.



✎ *The auxiliary array* `count` *will be created by the program and will store the number of grades for each student that are greater than 89.*

Obviously, to create the array `count`, the program will iterate through columns. The solution is as follows.

Project_33.4-2b

```
static final int STUDENTS = 10;
static final int LESSONS = 5;
public static void main(String[] args) throws Exception {
   int i, j;
   //Read names and grades all together. Iterate through columns in array grades
   String[] names = new String[STUDENTS];
   int[][] grades = new int[LESSONS][STUDENTS];
   for (j = 0; j <= STUDENTS - 1; j++) {
      System.out.print("Enter name for student No. " + (j + 1) + ": ");
      names[j] = cin.nextLine();
      for (i = 0; i <= LESSONS - 1; i++) {
         System.out.print("Enter grade No. " + (i + 1) + " for " + names[j] + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Create array count. Iterate through columns
   int[] count = new int[STUDENTS];
```

```
   for (j = 0; j <= STUDENTS - 1; j++) {
      count[j] = 0;
      for (i = 0; i <= LESSONS - 1; i++) {
         if (grades[i][j] > 89) {
            count[j]++;
         }
      }
   }
   //Displays the names of the students who have more than one grade greater than 89
   for (j = 0; j <= STUDENTS - 1; j++) {
      if (count[j] > 1) {
         System.out.println(names[j]);
      }
   }
}
```

Exercise 33.4-3 Using an Array Along with a HashMap

There are 30 students and each one of them has received their grades for a test. Write a Java program that prompts the user to enter the grades (as a letter) for each student. It then displays, for each student, the grade as a percentage according to the following table.

Grade	Percentage
A	90 - 100
B	80 - 89
C	70 - 79
D	60 - 69
E / F	0 - 59

Solution

A hashmap can be used to hold the given table. The solution is straightforward and requires no further explanation.

Project_33.4-3

```
static final int STUDENTS = 30;
public static void main(String[] args) throws Exception {
   int i;
   String grade, gradeAsPercentage;
   HashMap<String, String> gradesTable = new HashMap<>(
```

```
        Map.of("A", "90-100", "B", "80-89", "C", "70-79", "D", "60-69", "E", "0-59", "F",
        "0-59")
    );
    String[] names = new String[STUDENTS];
    String[] grades = new String[STUDENTS];
    for (i = 0; i <= STUDENTS - 1; i++) {
        System.out.print("Enter student name No" + (i + 1) + ": ");
        names[i] = cin.nextLine();
        System.out.print("Enter their grade: ");
        grades[i] = cin.nextLine();
    }
    for (i = 0; i <= STUDENTS - 1; i++) {
        grade = grades[i];
        gradeAsPercentage = gradesTable.get(grade);
        System.out.println(names[i] + " " + gradeAsPercentage);
    }
}
```

Now, if you fully understood how the last for-loop works, then take a look in the code fragment that follows. It is equivalent to that last for-loop, but it performs more efficiently, since it uses fewer variables!

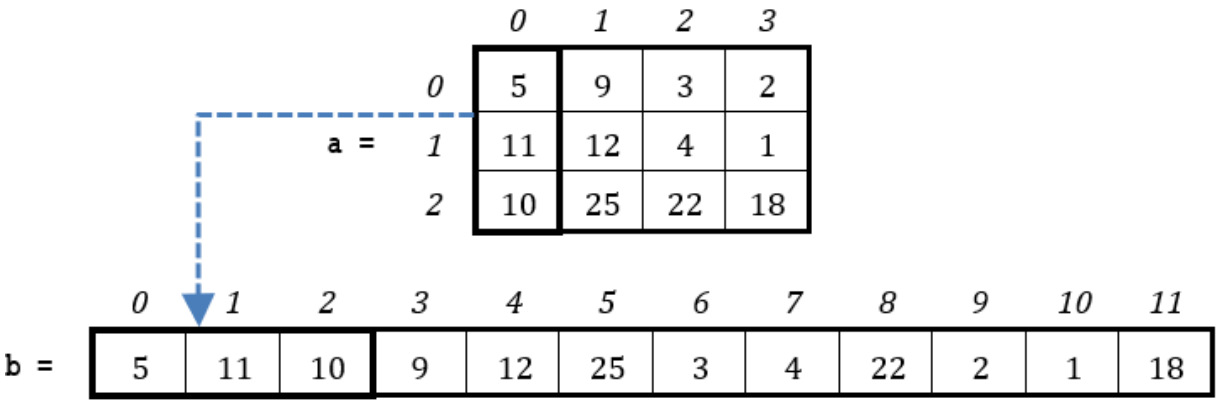
```
for (i = 0; i <= STUDENTS - 1; i++) {
    System.out.println(names[i] + " " + gradesTable.get(grades[i]));
}
```

33.5 Creating a One-Dimensional Array from a Two-Dimensional Array

To more easily understand how to create a one-dimensional array from a two-dimensional array, let's use an example.

Write a Java program that creates a one-dimensional array of 12 elements from an existing two-dimensional array of 3 × 4 (shown below), as follows: The elements of the first column of the two-dimensional array must be placed in the first three positions of the one-dimensional array, the elements of the second column must be placed in the next three positions, and so on.

The two-dimensional 3 × 4 array along with the new one-dimensional array are presented below.



The Java program that follows creates the new one-dimensional array, iterating through columns, as it is more convenient. It uses the existing array given in the example.

Project_33.5

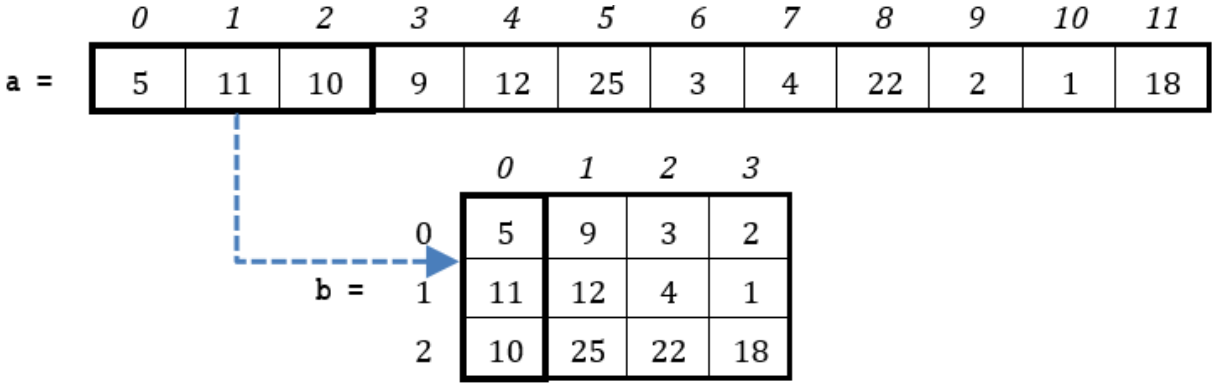
```
static final int ROWS = 3;
static final int COLUMNS = 4;
static final int ELEMENTS = ROWS * COLUMNS;
public static void main(String[] args) throws Exception {
   int i, j, k;
   int[][] a = {
      {5, 9, 3, 2},
      {11, 12, 4, 1},
      {10, 25, 22, 18}
   };
   k = 0; //This is the index of the new array.
   int[] b = new int[ROWS * COLUMNS];
   for (j = 0; j <= COLUMNS - 1; j++) { //Iterate through columns
      for (i = 0; i <= ROWS - 1; i++) {
         b[k] = a[i][j];
         k++;
      }
   }
   for (k = 0; k <= ELEMENTS - 1; k++) {
      System.out.print(b[k] + "\t");
   }
}
```

33.6 Creating a Two-Dimensional Array from a One-Dimensional Array

To more easily understand how to create a two-dimensional array from a one-dimensional array, let's use an example.

Write a Java program that creates a two-dimensional array of 3 × 4 from an existing one-dimensional array of 12 elements (*shown below*), *as follows: The first three elements of the one-dimensional array must be placed in the first column of the two-dimensional array, the next three elements of the one-dimensional array must be placed in the next column of the two-dimensional array, and so on.*

The one-dimensional array of 12 elements along with the new two-dimensional array are presented below.



The Java program that follows creates the new two-dimensional array, iterating through columns, which is more convenient. It uses the existing array given in the example.

Project_33.6

```
static final int ROWS = 3;
static final int COLUMNS = 4;
public static void main(String[] args) throws Exception {
   int k, j, i;
   int[] a = {5, 11, 10, 9, 12, 25, 3, 4, 22, 2, 1, 18};
   k = 0; //This is the index of array a.
   int[][] b = new int[ROWS][COLUMNS];
   for (j = 0; j <= COLUMNS - 1; j++) { //Iterate through columns
      for (i = 0; i <= ROWS - 1; i++) {
         b[i][j] = a[k];
         k++;
      }
   }
   for (i = 0; i <= ROWS - 1; i++) { //Iterate through rows
      for (j = 0; j <= COLUMNS - 1; j++) {
         System.out.print(b[i][j] + "\t");
      }
      System.out.println();
   }
}
```

33.7 Useful Data Structures Methods (Subprograms), and More

Counting the number of elements

```
array_name.length
```

The `length` property contains the number of elements of an array.

Example

Project_33.7a

```
public static void main(String[] args) throws Exception {
   int i;
   int[] a = {3, 6, 10, 12, 4, 2, 1};
   System.out.println(a.length); //It displays: 7
   for (i = 0; i <= a.length - 1; i++) {
      System.out.print(a[i] + " "); //It displays: 3 6 10 12 4 2 1
   }
   int[][] b = {
      {5, 9, 3, 2},
      {11, 12, 4, 1},
      {10, 25, 22, 18}
   };
   System.out.println(b.length); //It displays: 3
   System.out.println(b[0].length); //It displays: 4
}
```

✎ *In the two-dimensional array* `b`, `b.length` *contains the value of 3 (three rows), while* `b[0].length` *contains the value of 4 (four columns). If you want to get the total number of elements* ***in all the dimensions*** *of the array, you could use the product* `b.length * b[0].length`, *which would equal the value of 12.*

✎ *Contrary to the* `length()` *method that you learned in Section 14.3, here* `length` *is an attribute. Therefore, you must not put parentheses at the end. You will learn more about attributes in Part VIII.*

Sorting an array

```
Arrays.sort(array_name)
```

This method sorts the array `array_name` in ascending order.

✎ *Sorting is the process of putting the elements of an array in a certain order.*

Example

Project_33.7b

```
public static void main(String[] args) throws Exception {
   int i;
   int[] a = {3, 6, 10, 2, 1, 10, 12, 4};
   Arrays.sort(a);
   for (i = 0; i <= a.length - 1; i++) {
      System.out.print(a[i] + " "); //It displays: 1 2 3 4 6 10 10 12
   }
   String[] b = {"Hermes", "Apollo", "Dionysus"};
   Arrays.sort(b);
   for (i = 0; i <= b.length - 1; i++) {
      System.out.print(b[i] + " "); //It displays: Apollo Dionysus Hermes
   }
}
```

In order to use the `Arrays` *class in your project, you need to import the corresponding library using the* `import java.util.Arrays` *statement at the beginning of the project.*

Checking if Key Exists

*struct.***containsKey**(*key_name*)

This method returns `true` if the key *key_name* exists in the hashmap *struct*; it returns `false` otherwise.

Example

Project_33.7c

```
public static void main(String[] args) throws Exception {
   HashMap<String, String> family = new HashMap<>(
      Map.of("father", "John", "mother", "Maria", "son", "George", "daughter", "Helen")
   );
   if (family.containsKey("father")) {
      System.out.println(family.get("father")); //It displays: John
   }
   if (!family.containsKey("grandpa")) {
      System.out.println("No grandpa"); //It displays: No gradpa
   }
}
```

The statement `if (!family.containsKey("grandpa"))` *is equivalent to the statement* `if (family. containsKey("grandpa") == false)`.

33.8 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Processing each row individually means that every row is processed separately, and the result of each row can then be used individually for further processing.
2) The following code fragment displays the word “Okay” when the sum of the elements of each column is less than 100.

```
for (i = 0; i <= ROWS - 1; i++) {
   total = 0;
   for (j = 0; j <= COLUMNS - 1; j++) {
      total += a[i][j];
   }
   if (total < 100) System.out.println("Okay");
}
```

3) Processing each column individually means that every column is processed separately and the result of each row can be then used individually for further processing.
4) The following code fragment displays the sum of the elements of each column.

```
total = 0;
for (j = 0; j <= COLUMNS - 1; j++) {
   for (i = 0; i <= ROWS - 1; i++) {
      total += a[i][j];
   }
   System.out.println(total);
}
```

5) Suppose that there are 10 students and each one of them has received their grades for five lessons. Given this information, it is possible to design an array so that the rows refer to students and the columns refer to lessons, but not the other way around, that is, the rows referring to lessons and the columns referring to students.
6) A one-dimensional array can be created from a two-dimensional array, but not the opposite.
7) A one-dimensional array can be created from a three-dimensional array.
8) The following two code fragments display the same value.

```
int[] a = {1, 6, 12, 2, 1};
System.out.print(a.length);
String a = "Hello";
System.out.print(a.length());
```

9) The following code fragment displays three values.

```
int[] a = {10, 20, 30, 40, 50};
for (i = 3; i <= a.length - 1; i++) {
   System.out.println(a[i]);
}
```

10) The following code fragment displays the values of all elements of the array `b`.

```
int[] b = {10, 20, 30, 40, 50};
for (i = 0; i <= b.length - 1; i++) {
   System.out.println(i);
}
```

11) The following code fragment doubles the values of all elements of the array `b`.

```
for (i = 0; i < b.length; i++) {
   b[i] *= 2;
}
```

12) The following code fragment displays the smallest value of array `b`.

```
int[] b = {3, 6, 10, 2, 1, 12, 4};
Arrays.sort(b);
System.out.println(b[0]);
```

13) The following two code fragments display the greatest value of array `b`.

```
int[] b = {3, 6, 10, 2, 1, 12, 4};
Arrays.sort(b);
System.out.println(b[b.length - 1]);
```

```
int[] b = {3, 6, 10, 2, 1, 12, 4};
Arrays.sort(b);
System.out.println(b[6]);
```

33.9 Review Questions: Multiple Choice

Select the correct answer for each of the following statements.

1) The following code fragment

```
int[] total = new int[ROWS];
for (i = 0; i <= ROWS - 1; i++) {
   total[i] = 0;
   for (j = 0; j <= COLUMNS - 1; j++) {
      total[i] += a[i][j];
   }
   System.out.println(total[i]);
```

```
}
```

a) displays the sum of the elements of each row.
b) displays the sum of the elements of each column.
c) displays the sum of all the elements of the array.
d) none of the above

2) The following code fragment

```
for (j = 0; j <= COLUMNS - 1; j++) {
   total = 0;
   for (i = 0; i <= ROWS - 1; i++) {
      total += a[i][j];
   }
   System.out.println(total);
}
```

a) displays the sum of the elements of each row.
b) displays the sum of the elements of each column.
c) displays the sum of all the elements of the array.
d) none of the above

3) The following code fragment

```
total = 0;
for (j = 0; j <= COLUMNS - 1; j++) {
   for (i = 0; i <= ROWS - 1; i++) {
      total += a[i][j];
   }
   System.out.println(total);
}
```

a) displays the sum of the elements of each row.
b) displays the sum of the elements of each column.
c) displays the sum of all the elements of the array.
d) none of the above

4) The following code fragment

```
k = 0;
for (i = ROWS - 1; i >= 0; i++) {
   for (j = 0; j <= COLUMNS - 1; j--) {
      b[k++] = a[i][j];
   }
}
```

a) creates a one-dimensional array from a two-dimensional array.

b) creates a two-dimensional array from a one-dimensional array.

c) does not satisfy the property of definiteness

d) none of the above

5) The following code fragment

```
k = 0;
for (i = 0; i <= ROWS - 1; i++) {
   for (j = COLUMNS - 1; j >= 0; j--) {
      b[i][j] = a[k++];
   }
}
```

a) creates a one-dimensional array from a two-dimensional array.

b) creates a two-dimensional array from a one-dimensional array.

c) none of the above

6) The following two code fragments

```
int[] a = {3, 6, 10, 2, 4, 12, 1};
for (i = 0; i < 7; i++) {
   System.out.println(a[i]);
}
int[] a = {3, 6, 10, 2, 4, 12, 1};
for (i = 0; i <= a.length - 1; i++) {
   System.out.println(a[i]);
}
```

a) produce the same results.

b) do not produce the same results.

c) none of the above

7) The following two code fragments

```
int[] a = {3, 6, 10, 2, 4, 12, 1};
for (i = 0; i < a.length; i++) {
   System.out.println(a[i]);
}
int[] a = {3, 6, 10, 2, 4, 12, 1};
for (int element : a) {
   System.out.println(element);
}
```

a) produce the same results.

b) do not produce the same results.

c) none of the above

8) The following two code fragments

```
int[] a = {6, 4, 3, 2, 1};
Arrays.sort(a);
for (i = 0; i <= a.length - 1; i++) {
   System.out.println(a[i]);
}
int[] a = {1, 2, 3, 4, 6};
for (int x : a) {
   System.out.println(x);
}
```

a) produce the same results.

b) do not produce the same results.

c) none of the above

33.10 Review Exercises

Complete the following exercises.

1) There are 15 students and each one of them has received their grades for five tests. Write a Java program that lets the user enter the grades (as a percentage) for each student for all tests. It then calculates, for each student, the average grade and displays it as a letter grade according to the following table.

Grade	Percentage
A	90 - 100
B	80 - 89
C	70 - 79
D	60 - 69
E / F	0 - 59

2) On Earth, a free-falling object has an acceleration of 9.81 m/s^2 downward. This value is denoted by *g*. A student wants to calculate that value using an experiment. She allows five different objects to fall downward from a known height and measures the time they need to reach the floor. She does this 10 times for each object. Then, using a formula she calculates *g* for each object, for each fall. But since her chronometer is not so accurate, she needs a Java program that lets her

enter all calculated values of g in a 5×10 array and then, it calculates and displays

a) for each object, the average value of g
b) for each fall, the average value of g
c) the overall average value of g

3) A basketball team with 15 players plays 12 matches. Write a Java program that lets the user enter, for each player, the number of points scored in each match. The program must then display
   a) for each player, the total number of points scored
   b) for each match, the total number of points scored
4) Write a Java program that lets the user enter the hourly measured temperatures of 20 cities for a period of one day, and then displays the hours in which the average temperature of all the cities was below 10 degrees Fahrenheit.
5) In a football tournament, a football team with 24 players plays 10 matches. Write a Java program that lets the user enter, for each player, a name as well as the number of goals they scored in each match. The program must then display
   a) for each player, his name and the average number of goals he scored
   b) for each match, the index number of the match (1, 2, 3, and so on) and the total number of goals scored
6) There are 12 students and each one of them has received their grades for six lessons. Write a Java program that lets the user enter the name of the student as well as their grades in all lessons and then displays
   a) for each student, their name and average grade
   b) for each lesson, the average grade
   c) the names of the students who have an average grade less than 60
   d) the names of the students who have an average grade greater than 89, and the message “Bravo!” next to it

   Assume that the user enters valid values between 0 and 100.
7) In a song contest, each artist sings a song of their choice. There are five judges and 15 artists, each of whom is scored for their performance. Write a Java program that prompts the user to enter the names of the

judges, the names of the artists, the title of the song that each artist sings, and the score they get from each judge. The program must then display

a) for each artist, their name, the title of the song, and their total score
b) for each judge, their name and the average value of the score they gave

8) The Body Mass Index (BMI) is often used to determine whether a person is overweight or underweight for their height. The formula used to calculate BMI is

$$BMI = \frac{weight \cdot 703}{height^2}$$

Write a Java program that lets the user enter into two arrays the weight (in pounds) and height (in inches) of 30 people, measured on a monthly basis, for a period of one year (January to December). The program must then calculate and display

a) for each person, their average weight, average height, and average BMI
b) for each person, their BMI in May and in August

Please note that all people are adults but some of them are between the ages of 18 and 25. This means they may still grow taller, thus their height might be different each month!

9) Write a Java program that lets the user enter the electric meter reading in kilowatt-hours (kWh) at the beginning and at the end of a month for 1000 consumers. The program must then calculate and display

a) for each consumer, the amount of kWh consumed and the amount of money that must be paid given a cost of each kWh of $0.07 and a value added tax (VAT) rate of 19%
b) the total consumption and the total amount of money that must be paid.

10) Write a Java program that prompts the user to enter an amount in US dollars and calculates and displays the corresponding currency value in Euros, British Pounds Sterling, Australian Dollars, and Canadian Dollars. The tables below contain the exchange rates for each currency for a period of five working days. The program must calculate the

average value of each currency and do the conversions based on that average value.

currency =

British Pound Sterling
Euro
Canadian Dollar
Australian Dollar

rate =

1.420	1.421	1.432	1.431	1.441
1.043	1.056	1.038	1.022	1.029
0.757	0.764	0.760	0.750	0.749
0.620	0.625	0.629	0.636	0.639

11) Gross pay depends on the pay rate and the total number of hours worked per week. However, if someone works more than 40 hours, they get paid time-and-a-half for all hours worked over 40. Write a Java program that lets the user enter a pay rate, as well as the names of 10 employees and the number of hours that they worked each day (Monday to Friday). The program must then calculate and display
   a) the names of employees who worked overtime
   b) for each employee, their name and the average daily gross pay
   c) the total gross pay for all employees
   d) for each employee, their name, the name of the day they worked overtime (more than 8 hours), and the message “Overtime!”
   e) for each day, the name of the day and the total gross pay

12) Write a Java program to create a one-dimensional array of 12 elements from the two-dimensional array shown below, as follows: the first row of the two-dimensional array must be placed in the first four positions of the one-dimensional array, the second row of the two-dimensional array must be placed in the next four positions of the one-dimensional array, and the last row of the two-dimensional array must be placed in the last four positions of the one-dimensional array.

a =

9	9	2	6
4	1	10	11
12	15	7	3

13) Write a Java program to create a 3 × 3 array from the one-dimensional array shown below, as follows: the first three elements of the one-dimensional array must be placed in the last row of the two-

dimensional array, the next three elements of the one-dimensional array must be placed in the second row of the two-dimensional array, and the last three elements of the one-dimensional array must be placed in the first row of the two-dimensional array.

a =

16	12	3	5	6	9	18	19	20

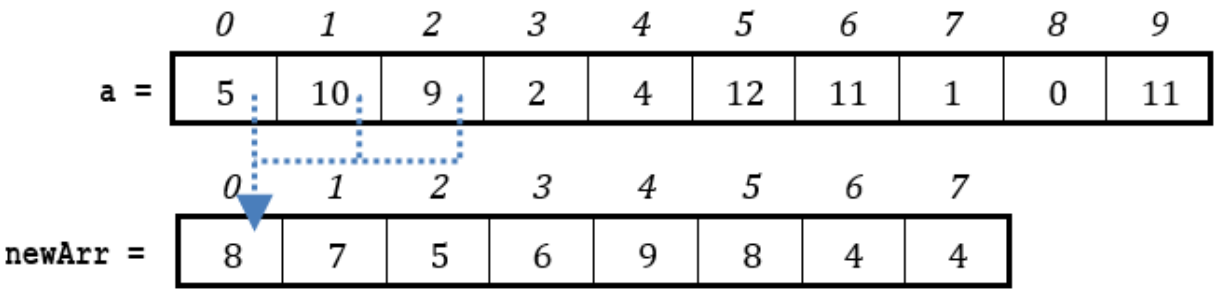
Chapter 34
More with Data Structures

34.1 Simple Exercises with Arrays

Exercise 34.1-1 Creating an Array that Contains the Average Values of its Neighboring Elements

Write a Java program that lets the user enter 100 positive numerical values into an array. Then, the program must create a new array of 98 elements. This new array must contain, in each position the average value of the three elements that exist in the current and the next two positions of the user-provided array.

Solution Let's try to understand this exercise through an example using 10 elements.



Array `newArr` is the new array that is created. In array `newArr`, the element at position 0 is the average value of the elements in the current and the next two positions of array `a`; that is, (5 + 10 + 9) / 3 = 8. The element at position 1 is the average value of the elements in the current and the next two positions of array `a`; that is, (10 + 9 + 2) / 3 = 7, and so on.

The Java program is as follows.

Project_34.1-1

```
static final int ELEMENTS_OF_A = 100; static final int ELEMENTS_OF_NEW = ELEMENTS_OF_A -
2;
public static void main(String[] args) throws Exception {
   int i;
   double[] a = new double[ELEMENTS_OF_A]; for (i = 0; i <= ELEMENTS_OF_A - 1; i++) {
      a[i] = Double.parseDouble(cin.nextLine());
   }
   double[] newArr = new double[ELEMENTS_OF_NEW]; for (i = 0; i <= ELEMENTS_OF_NEW - 1;
   i++) {
      newArr[i] = (a[i] + a[i + 1] + a[i + 2]) / 3;
```

```
    }
    for (i = 0; i <= ELEMENTS_OF_NEW - 1; i++) {
       System.out.print(newArr[i] + "\t");
    }
}
```

Exercise 34.1-2 Creating an Array with the Greatest Values

Write a Java program that lets the user enter numerical values into arrays `a` *and* `b` *of 20 elements each. Then, the program must create a new array* `newArr` *of 20 elements. The new array must contain in each position the greatest value of arrays* `a` *and* `b` *of the corresponding position.*

Solution Nothing new here! You need two for-loops to read the values for arrays `a` and `b`, one for creating the array `newArr`, and one to display the array `newArr` on the screen.

The Java program is shown here.

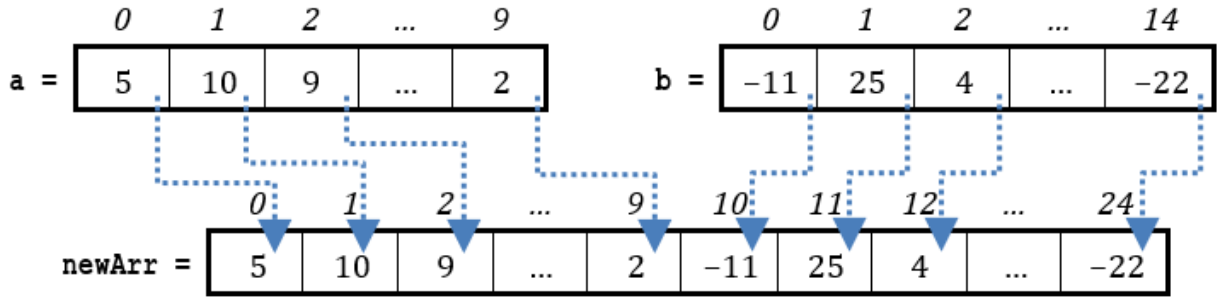
Project_34.1-2

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
   int i;
   //Read arrays a and b double[] a = new double[ELEMENTS]; for (i = 0; i <= ELEMENTS -
   1; i++) {
      a[i] = Double.parseDouble(cin.nextLine());
   }
   double[] b = new double[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      b[i] = Double.parseDouble(cin.nextLine());
   }
   //Create array newArr double[] newArr = new double[ELEMENTS]; for (i = 0; i <=
   ELEMENTS - 1; i++) {
      if (a[i] > b[i]) {
         newArr[i] = a[i];
      }
      else {
         newArr[i] = b[i];
      }
   }
   //Display array newArr for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.println(newArr[i]);
   }
}
```

Exercise 34.1-3 Merging One-Dimensional Arrays

Write a Java program that, for two given arrays `a` *and* `b` *of 10 and 15 elements, respectively, creates a new array* `newArr` *of 25 elements. This new array must contain in the first 10 positions the elements of array* `a`*, and in the next 15 positions the elements of array* `b`*.*

Solution As you can see in the example presented next, there is a direct one-to-one correspondence between the index positions of the elements of array a and those of array newArr. Specifically, the element from position 0 of array a is stored in position 0 of array newArr, the element from position 1 of array a is stored in position 1 of array newArr, and so forth. However, this correspondence doesn't hold for array b; its elements need to be shifted by 10 positions in newArr.



In order to assign the values of array a to array `newArr` you can use the following code fragment.

```
for (i = 0; i <= a.length - 1; i++) {
   newArr[i] = a[i]; }
```

However, to assign the values of array `b` to array `newArr` your code fragment should be slightly different as shown here.

```
for (i = 0; i <= b.length - 1; i++) {
   newArr[a.length + i] = b[i]; }
```

The final Java program is as follows.

Project_34.1-3

```
public static void main(String[] args) throws Exception {
   int i;
   //Create arrays a and b int[] a = {5, 10, 9, 6, 7, -6, 13, 12, 11, 2}; int[] b =
   {-11, 25, 4, 45, 67, 87, 34, 23, 33, 55, 13, 15, -4, -2, -22};
   //Create array newArr int[] newArr = new int[a.length + b.length]; for (i = 0; i <=
   a.length - 1; i++) {
      newArr[i] = a[i];
   }
   for (i = 0; i <= b.length - 1; i++) {
      newArr[a.length + i] = b[i];
```

```
    }
    //Display array newArr for (i = 0; i <= newArr.length - 1; i++) {
       System.out.print(newArr[i] + "\t");
    }
}
```

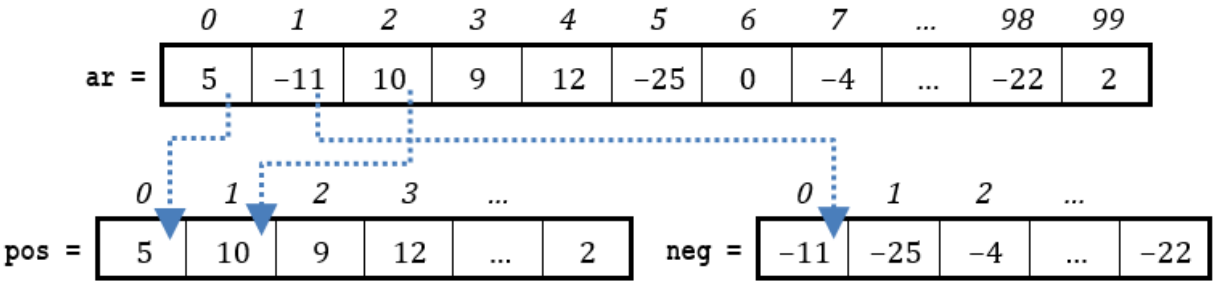
The `length` *attribute contains the total number of the elements in an array.*

Contrary to the `length()` *method that you learned in Section 14.3, here* `length` *is an attribute. Therefore, you must not put parentheses at the end. You will learn more about attributes in Part VIII.*

Exercise 34.1-4 Creating Two Arrays – Separating Positive from Negative Values

Write a Java program that lets the user enter 100 numerical values into an array and then creates two new arrays, `pos` *and* `neg`*. Array* `pos` *must contain positive values, whereas array* `neg` *must contain the negative ones. The value 0 (if any) must not be added to either of the final arrays,* `pos` *or* `neg`*.*

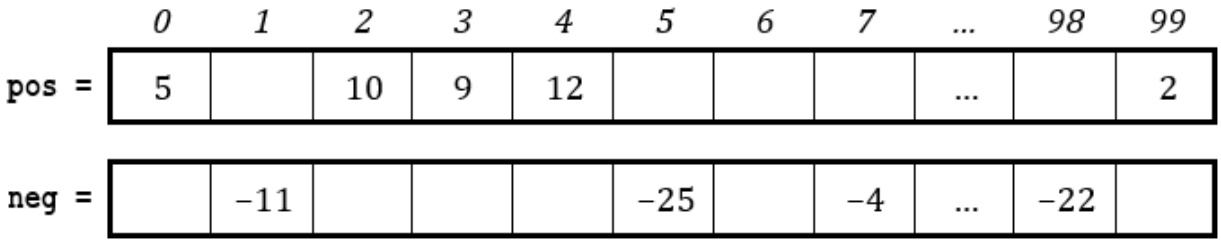
Solution Let's analyze this approach using the following example.



In this exercise, there is no one-to-one correspondence between the index positions of the elements of array `ar` and the arrays `pos` and `neg`. For example, the element from position 1 of array `ar` is not stored in position 1 of array `neg`, or the element from position 2 of array `ar` is not stored in position 2 of array `pos`. Thus, you **cannot** do the following,

```
for (i = 0; i <= ELEMENTS - 1; i++) {
    if (ar[i] > 0) {
        pos[i] = ar[i];
    }
    else if (ar[i] < 0) {
        neg[i] = ar[i];
    }
}
```

because it will result in the following two arrays.



What you need here are two independent index variables: `posIndex` for the array `pos`, and `negIndex` for the array `neg`. These index variables must be incremented independently, and only when an element is added to the corresponding array. The index variable `posIndex` must be incremented only when an element is added to the array `pos`, and the index variable `negIndex` must be incremented only when an element is added to the array `neg`, as shown in the code fragment that follows.

```
posIndex = 0;
negIndex = 0;
for (i = 0; i <= ELEMENTS - 1; i++) {
   if (ar[i] > 0) {
      pos[posIndex] = ar[i];
      posIndex++;
   }
   else if (ar[i] < 0) {
      neg[negIndex] = ar[i];
      negIndex++;
   }
}
```

which can equivalently be written as

```
posIndex = 0;
negIndex = 0;
for (i = 0; i <= ELEMENTS - 1; i++) {
      if (ar[i] > 0) {
            pos[posIndex++] = ar[i];
      }
      else if (ar[i] < 0) {
            neg[negIndex++] = ar[i];
      }
}
```

✎ *Note that variables* `posIndex` *and* `negIndex` *have dual roles. When the loop iterates, each points to the next position in which a new element must be*

placed. But when the loop finishes iterating, variables `posIndex` *and* `negIndex` *also contain the total number of elements in each corresponding array!*

The complete solution is presented next.

Project_34.1-4

```
static final int ELEMENTS = 100;
public static void main(String[] args) throws Exception {
   int i, posIndex, negIndex;
   double[] ar = new double[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      ar[i] = Double.parseDouble(cin.nextLine());
   }
   //Create arrays pos and neg posIndex = 0; negIndex = 0; double[] pos = new
   double[ELEMENTS]; double[] neg = new double[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1;
   i++) {
      if (ar[i] > 0) {
         pos[posIndex++] = ar[i];
      }
      else if (ar[i] < 0) {
         neg[negIndex++] = ar[i];
      }
   }
   for (i = 0; i <= posIndex - 1; i++) {
      System.out.print(pos[i] + "\t");
   }
   System.out.println(); for (i = 0; i <= negIndex - 1; i++) {
      System.out.print(neg[i] + "\t");
   }
}
```

✎ *Note that the arrays* `pos` *and* `neg` *contain a total number of* `posIndex` *and* `negIndex` *elements respectively. This is why the two last loop control structures iterate until variable* `i` *reaches values* `posIndex - 1` *and* `negIndex - 1`*, respectively, and not until* `ELEMENTS - 1`*, as you may mistakenly expect.*

Exercise 34.1-5 Creating an Array with Those who Contain Digit 5

Write a Java program that lets the user enter 100 two-digit integers into an array and then creates a new array of only the integers that contain at least one of the digit 5.

Solution This exercise requires some knowledge from the past. In Exercise 13.1-2 you learned how to use the quotient and the remainder to split an integer into its individual digits. Here, the user-provided integers have two

digits; therefore, you can use the following code fragment to split any two-digit integer contained in variable x.

```
lastDigit = x % 10;
firstDigit = (int)(x / 10);
```

The program that follows uses an extra variable as an index for the new array. This is necessary when you want to create a new array using values from an old array and there is no one-to-one correspondence between their index positions. Of course, this variable must increase by 1 only when a new element is added into the new array. Moreover, when the loop that creates the new array finishes iterating, the value of this variable also matches the total number of elements in the new array! The final Java program is as follows.

Project_34.1-5

```
static final int ELEMENTS = 100;
public static void main(String[] args) throws Exception {
   int i, k, lastDigit, firstDigit;
   int[] a = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      a[i] = Integer.parseInt(cin.nextLine());
   }
   k = 0;
   int[] b = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      lastDigit = a[i] % 10;
      firstDigit = (int)(a[i] / 10);
      if (lastDigit == 5 || firstDigit == 5) {
         b[k++] = a[i];
      }
   }
   for (i = 0; i <= k - 1; i++) {
      System.out.print(b[i] + "\t");
   }
}
```

34.2 Data Validation with Arrays

As you have already been taught in Section 30.3, there are three approaches that you can use to validate data input. Your approach will depend on whether or not you wish to display an error message, and whether you wish to display a different error message for each type of input error or just a generic error message for any kind of error. Let's see how those three approaches can be adapted and used with arrays.

First approach – Validating data input without error messages In Section 30.3, you learned how to validate one single value entered by the user without displaying any error messages. For your convenience, the code fragment given in general form is presented once again.

```
do {
   System.out.print("Prompt message"); input_data = cin.nextLine(); } while (input_data
   test 1 fails || input_data test 2 fails || …);
```

Do you remember how this operates? If the user enters a valid value, the flow of execution simply proceeds to the next section of the program. However, if they enter an invalid value, the primary objective is to repeatedly prompt them until they eventually provide a valid one.

You can use the same principle when entering data into arrays. If you use a for-loop to iterate for all elements of the array, the code fragment becomes as follows.

```
for (i = 0; i <= ELEMENTS - 1; i++) {
   do {
      System.out.print("Prompt message");
      input_data = cin.nextLine();
   } while (input_data test 1 fails || input_data test 2 fails || …); input_array[i] =
   input_data; }
```

As you can see, when the flow of execution exits the nested post-test loop structure, the variable *input_data* definitely contains a valid value which in turn is assigned to an element of the array *input_array*. However, the same process can be implemented more simply, without using the extra variable *input_data*, as follows.

```
for (i = 0; i <= ELEMENTS - 1; i++) {
   do {
      System.out.print("Prompt message");
      input_array[i] = cin.nextLine();
   } while (input_array[i] test 1 fails || input_array[i] test 2 fails || … ); }
```

Second approach – Validating data input with a generic error message As before, the next code fragment is taken from Section 30.3 and adapted to operate with an array. It validates data input and displays a generic error message (that is, the same error message for any type of input error).

```
for (i = 0; i <= ELEMENTS - 1; i++) {
   System.out.print("Prompt message"); input_array[i] = cin.nextLine(); while
   (input_array[i] test 1 fails || input_array[i] test 2 fails || … ) {
      System.out.println("Error message");
      System.out.print("Prompt message");
      input_array[i] = cin.nextLine();
```

```
   }
}
```

Third approach – Validating data input with different error messages Once again, the next code fragment is taken from Section 30.3 and adapted to operate with an array. It validates data input and displays a different error message for each type of input error.

```
for (i = 0; i <= ELEMENTS - 1; i++) {
   do {
      System.out.print("Prompt message");
      input_array[i] = cin.nextLine();
      failure = false;
      if (input_array[i] test 1 fails) {
         System.out.println("Error message 1");
         failure = true;
      }
      else if (input_array[i] test 2 fails) {
         System.out.println("Error message 2");
         failure = true;
      }
      else if (…
      …
      }
   } while (failure); }
```

Exercise 34.2-1 Displaying Odds in Reverse Order

Write a Java program that prompts the user to enter 20 odd positive integers into an array and then displays them in the exact reverse of the order in which they were provided. The program must validate data input, preventing the user from entering a non-positive value, a float, or an even integer. Solve this exercise in three versions: a) Validate data input without displaying any error messages.

b) Validate data input and display a generic error message.

c) Validate data input and display a different error message for each type of input error.

Solution All three approaches for validating data input that you learned in Section 34.2 will be presented here. Let's first solve this exercise without data validation.

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
   int i, x;
```

```
   int[] odds = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print("Enter an odd positive integer: "); [More…]
      x = Integer.parseInt(cin.nextLine());
      odds[i] = x;
   }
   //Display elements backwards for (i = ELEMENTS - 1; i >= 0; i--) {
      System.out.print(odds[i] + "\t");
   }
}
```

Validation without error messages To validate data input without displaying any error messages, use the first approach from Section 34.2. Simply replace the statements marked with a dashed rectangle with the following code fragment.

```
do {
   System.out.print("Enter an odd positive integer: "); x =
   Double.parseDouble(cin.nextLine()); } while (x <= 0 || x != (int)x || x % 2 == 0);
   odds[i] = (int)x;
```

The final program becomes Project_34.2-1a

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
    int i; double x;
    int[] odds = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++)
    {
        do { [More…]
            System.out.print("Enter an odd positive integer: ");
            x = Double.parseDouble(cin.nextLine());
        } while (x <= 0 || x != (int)x || x % 2 == 0);
        odds[i] = (int)x;
    }
    //Display elements backwards for (i = ELEMENTS - 1; i >= 0; i--) {
        System.out.print(odds[i] + "\t");
    }
}
```

Variable `x` must be of type `double`. This is necessary in order to allow the user to enter either an integer or a float (real).

Validation with a generic error message To validate data input and display a generic error message, replace the statements marked with the

dashed rectangle with a code fragment based on the second approach from Section 34.2. The Java program is as follows.

```
Project_34.2-1b
        static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
                int i; double x;
    int[] odds = new int[ELEMENTS]; for (i = 0; i <=
                ELEMENTS - 1; i++) {
      System.out.print("Enter an odd positive integer:
                "); [More…]
        x = Double.parseDouble(cin.nextLine());
      while (x <= 0 || x != (int)x || x % 2 == 0) {
          System.out.println("Invalid value!");
      System.out.print("Enter an odd positive integer:
                            ");
          x = Double.parseDouble(cin.nextLine());
                            }
                    odds[i] = (int)x;
                            }
  //Display elements backwards for (i = ELEMENTS - 1; i
                    >= 0; i--) {
            System.out.print(odds[i] + "\t");
                            }
                          }
```

Validation with different error messages Here, the replacing code fragment is based on the third approach from Section Section 34.2. To validate data input and display a different error message for each type of input error, the Java program is as follows.

```
Project_34.2-1c
        static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
            int i; boolean failure; double x;
             int[] odds = new int[ELEMENTS];
          for (i = 0; i <= ELEMENTS - 1; i++) {
```

```
                do {  [More…]
        System.out.print("Enter an odd positive integer:
                                ");
            x = Double.parseDouble(cin.nextLine());
                        failure = false;
                          if (x <= 0) {
             System.out.println("Invalid value: Non-
                        positive entered!");
                           failure = true;
                                }
                      else if (x != (int)x) {
             System.out.println("Invalid value: Float
                          entered!");
                           failure = true;
                                }
                      else if (x % 2 == 0) {
             System.out.println("Invalid value: Even
                          entered!");
                           failure = true;
                                }
                    } while (failure);
                    odds[i] = (int)x;
                                }
    //Display elements backwards for (i = ELEMENTS - 1; i
                        >= 0; i--) {
                System.out.print(odds[i] + "\t");
                                }
                               }
```

34.3 Finding Minimum and Maximum Values in Arrays

This is the third and last time that this subject is brought up in this book. The first time was in Section 23.2 using decision control structures and the second time was in Section 30.4 using loop control structures. So, there is not much left to discuss except the fact that when you want to find the minimum or maximum value of a data structure that already contains some values, you needn't worry about the initial values of variables `minimum` or

`maximum` because you can just assign to them the value of the first element of the data structure!

Exercise 34.3-1 Which Depth is the Greatest?

Write a Java program that lets the user enter the depths of 20 lakes and then displays the depth of the deepest one.

Solution After the user enters the depths of the 20 lakes in the array `depths`, the initial value of variable `maximum` can be set to the value of `depths[0]`, that is, the first element of array `depths`. The program can then search any subsequent value greater than this, starting from index 1. The final solution is quite straightforward and is presented next without further explanation.

Project_34.3-1

```
static final int LAKES = 20;
public static void main(String[] args) throws Exception {
   int i; double maximum;
   double[] depths = new double[LAKES]; for (i = 0; i <= LAKES - 1; i++) {
      depths[i] = Double.parseDouble(cin.nextLine());
   }
   maximum = depths[0]; //Initial value
   //Search thereafter, starting from index 1
   for (i = 1; i <= LAKES - 1; i++) {
      if (depths[i] > maximum) {
         maximum = depths[i];
      }
   }
   System.out.println(maximum); }
```

✎ *It wouldn't be wrong to start iterating from position 0 instead of 1, though the program would perform one useless iteration.*

✎ *It wouldn't be wrong to assign an "almost arbitrary" initial value to variable `maximum` but there is no reason to do so. The value of the first element is just fine! If you insist though, you can assign an initial value of 0, since there is no lake on planet Earth with a negative depth.*

Exercise 34.3-2 Which Lake is the Deepest?

Write a Java program that lets the user enter the names and the depths of 20 lakes and then displays the name of the deepest one.

Solution If you don't know how to find the name of the deepest lake, you may need to refresh your memory by re-reading Exercise 30.4-2.

In this exercise, you need two one-dimensional arrays: one to hold the names, and one to hold the depths of the lakes. The solution is presented next.

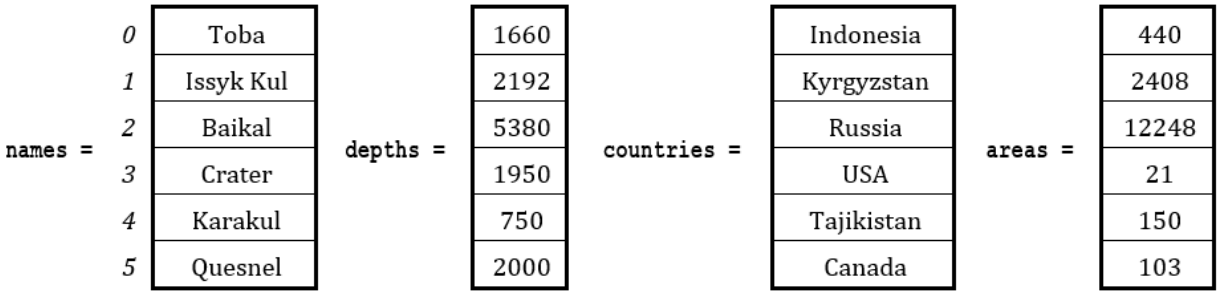
Project_34.3-2

```
static final int LAKES = 20;
public static void main(String[] args) throws Exception {
   int i; double maximum; String mName;
   String[] names = new String[LAKES]; double[] depths = new double[LAKES]; for (i = 0;
   i <= LAKES - 1; i++) {
      names[i] = cin.nextLine();
      depths[i] = Double.parseDouble(cin.nextLine());
   }
   maximum = depths[0]; mName = names[0]; for (i = 1; i <= LAKES - 1; i++) {
      if (depths[i] > maximum) {
         maximum = depths[i];
         mName = names[i];
      }
   }
   System.out.println(mName); }
```

Exercise 34.3-3 Which Lake, in Which Country, Having Which Average Area, is the Deepest?

Write a Java program that lets the user enter the names and the depths of 20 lakes as well as the country in which they belong, and their average area. The program must then display all available information about the deepest lake.

Solution Let's look at the next example of six lakes. The depths are expressed in feet and the average areas in square miles.



	names	depths	countries	areas
0	Toba	1660	Indonesia	440
1	Issyk Kul	2192	Kyrgyzstan	2408
2	Baikal	5380	Russia	12248
3	Crater	1950	USA	21
4	Karakul	750	Tajikistan	150
5	Quesnel	2000	Canada	103

It's evident that Lake Baikal holds the record as the deepest lake, positioned at index 2. If you were to approach this exercise in a manner similar to the previous exercise (Exercise 34.3-2), you would need three more variables to keep the name, country, and area each time a depth greater than the

previously stored one is found. However, the solution presented below employs a more efficient approach, using only **one** variable (`indexOfMax`) to track the index where these values are located.

Project_34.3-3

```
static final int LAKES = 20;
public static void main(String[] args) throws Exception {
   int i, indexOfMax; double maximum;
   String[] names = new String[LAKES]; double[] depths = new double[LAKES]; String[]
   countries = new String[LAKES]; double[] areas = new double[LAKES]; for (i = 0; i <=
   LAKES - 1; i++) {
      names[i] = cin.nextLine();
      depths[i] = Double.parseDouble(cin.nextLine());
      countries[i] = cin.nextLine();
      areas[i] = Double.parseDouble(cin.nextLine());
   }
   //Find the maximum depth and the index in which this maximum depth exists maximum =
   depths[0]; indexOfMax = 0; for (i = 1; i <= LAKES - 1; i++) {
      if (depths[i] > maximum) {
         maximum = depths[i];
         indexOfMax = i;
      }
   }
   //Display information using indexOfMax as index System.out.print(depths[indexOfMax] +
   " " + names[indexOfMax] + " "); System.out.println(countries[indexOfMax] + " " +
   areas[indexOfMax]); }
```

✎ *Assigning an initial value of 0 to variable* `indexOfMax` *is necessary since there is always a possibility that the maximum value does exist in position 0.*

Exercise 34.3-4 Which Students Have got the Greatest Grade?

Write a Java program that prompts the user to enter the names and the grades of 200 students and then displays the names of all those who share the one greatest grade. Using a loop control structure, the program must also validate data input and display an error message when the user enters an empty name or any negative values or values greater than 100 for grades.

Solution In this exercise, you need to validate both the names and the grades. A code fragment, given in general form, shows the data input stage.

```
static final int STUDENTS = 200;
public static void main(String[] args) throws Exception {
   String[] names = new String[STUDENTS]; int[] grades = new int[STUDENTS]; for (i = 0;
   i <= STUDENTS - 1; i++) {
```

```
    Prompt the user to enter a name and validate
    it. It cannot be empty!

    Prompt the user to enter a grade and validate
    it. It cannot be negative or greater than 100.
    }
…
```

After data input stage, a loop control structure must search for the greatest value, and then, another loop control structure must search the array `grades` for all values that are equal to that greatest value.

The solution in presented next.

Project_34.3-4

```
static final int STUDENTS = 200;
public static void main(String[] args) throws Exception {
   int i, maximum;
   String[] names = new String[STUDENTS]; int[] grades = new int[STUDENTS]; for (i = 0;
   i <= STUDENTS - 1; i++) {
      //Prompt the user to enter a name and validate it.
      System.out.print("Enter name for student No " + (i + 1) + ": ");
      names[i] = cin.nextLine();
      while (names[i].equals("")) {
         System.out.println("Error! Name cannot be empty!");
         System.out.print("Enter name for student No " + (i + 1) + ": ");
         names[i] = cin.nextLine();
      }
      //Prompt the user to enter a grade and validate it.
      System.out.print("Enter their grade: ");
      grades[i] = Integer.parseInt(cin.nextLine());
      while (grades[i] < 0 || grades[i] > 100) {
         System.out.println("Invalid value!");
         System.out.print("Enter their grade: ");
         grades[i] = Integer.parseInt(cin.nextLine());
      }
   }
   //Find the greatest grade maximum = grades[0]; for (i = 1; i <= STUDENTS - 1; i++) {
      if (grades[i] > maximum) {
         maximum = grades[i];
      }
   }
   //Displays the names of all those who share the one greatest grade
   System.out.println("The following students have got the greatest grade:"); for (i =
   0; i <= STUDENTS - 1; i++) {
      if (grades[i] == maximum) {
```

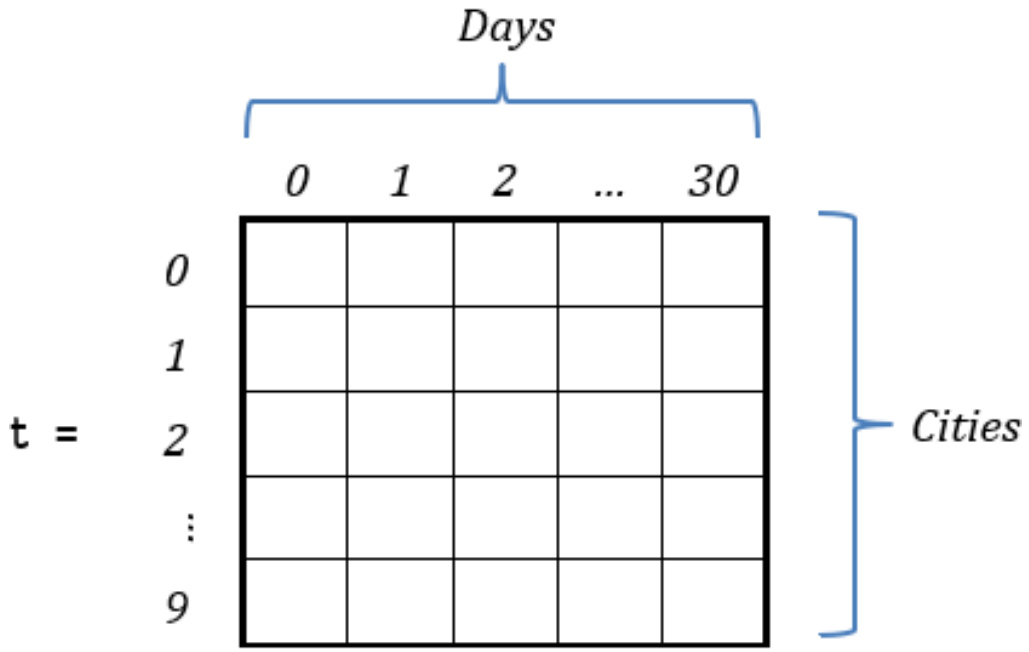
```
            System.out.println(names[i]);
        }
    }
}
```

✎ *Note that this exercise could not have been solved without the use of an array.*

Exercise 34.3-5 Finding the Minimum Value of a Two-Dimensional Array

Write a Java program that lets the user enter the temperatures (in degrees Fahrenheit) recorded at the same hour each day in January in 10 different cities. The Java program must display the lowest temperature.

Solution In this exercise, you need the following array.



✎ *The array* `t` *has 31 columns (0 to 30), as many as there are days in January.*

There is nothing new here. The initial value of variable `minimum` can be the value of the element `t[0][0]`. Then, the program can iterate through rows, or even through columns, to search for the minimum value. The solution is presented next.

🗀 Project_34.3-5

```
static final int CITIES = 10; static final int DAYS = 31;
public static void main(String[] args) throws Exception {
```

```
int i, j, minimum;
//Read array t int[][] t = new int[CITIES][DAYS]; for (i = 0; i <= CITIES - 1; i++) {
   for (j = 0; j <= DAYS - 1; j++) {
      t[i][j] = Integer.parseInt(cin.nextLine());
   }
}
//Find minimum minimum = t[0][0]; for (i = 0; i <= CITIES - 1; i++) {
   for (j = 0; j <= DAYS - 1; j++) {
      if (t[i][j] < minimum) {
         minimum = t[i][j];
      }
   }
}
System.out.println(minimum); }
```

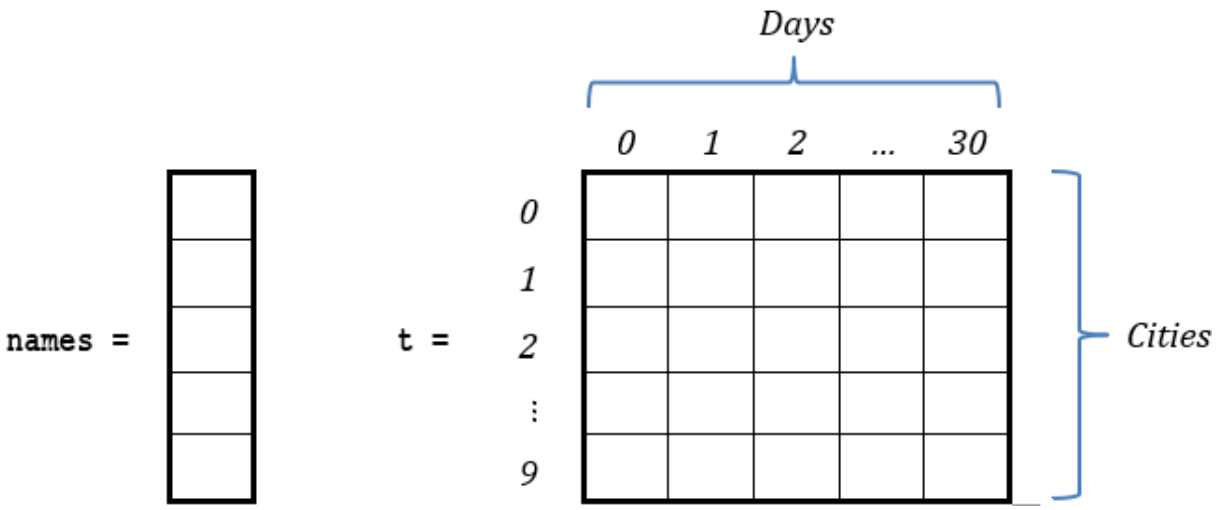
✎ *In this exercise you can**not** do the following because if you do, and variable* `j` *starts from 1, the whole column with index 0 won't be checked!*

```
//Find minimum minimum = t[0][0];
for (i = 0; i <= CITIES - 1; i++) {
   for (j = 1; j <= DAYS - 1; j++) { //This is wrong! Variable j must start from 0
      if (t[i][j] < minimum) {
         minimum = t[i][j];
      }
   }
}
```

Exercise 34.3-6 Finding the City with the Coldest Day

Write a Java program that lets the user enter the names of 10 cities as well as the temperatures (in degrees Fahrenheit) recorded at the same hour each day in January in those cities. The Java program must display the name of the city that had the lowest temperature and on which day it was recorded.

Solution In this exercise, the following two arrays are needed.



The solution is simple. Every time variable `minimum` updates its value, two variables, `m_i` and `m_j`, can hold the current values of variables `i` and `j` respectively. In the end, these two variables will contain the row index and the column index of the position in which the minimum value exists. The solution is as follows.

Project_34.3-6

```
static final int CITIES = 10; static final int DAYS = 31;
public static void main(String[] args) throws Exception {
   int i, j, minimum, m_i, m_j;
   String[] names = new String[CITIES]; int[][] t = new int[CITIES][DAYS]; for (i = 0; i
   <= CITIES - 1; i++) {
      names[i] = cin.nextLine();
      for (j = 0; j <= DAYS - 1; j++) {
         t[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   minimum = t[0][0]; m_i = 0;
   m_j = 0;
   for (i = 0; i <= CITIES - 1; i++) {
      for (j = 0; j <= DAYS - 1; j++) {
         if (t[i][j] < minimum) {
            minimum = t[i][j];
            m_i = i;
            m_j = j;
         }
      }
   }
   System.out.println("Minimum temperature: " + minimum); System.out.println("City: " +
   names[m_i]); System.out.println("Day: " + (m_j + 1)); }
```

✎ *Assigning an initial value of 0 to variables* `m_i` *and* `m_j` *is necessary since there is always a possibility that the minimum value is the value of the element* `t[0][0]`.

Exercise 34.3-7 Finding the Minimum and the Maximum Value of Each Row

Write a Java program that lets the user enter values into array `b` *of 20 × 30 elements and then finds and displays the minimum and the maximum values of each row.*

Solution There are two approaches, actually. The first approach creates two auxiliary one-dimensional arrays, `minimum` ***and*** `maximum`***, and then displays them. Arrays*** `minimum` ***and*** `maximum` ***will contain, in each position, the minimum and the maximum values of each row respectively. On the other hand, the second approach finds and directly displays the minimum and maximum values of each row. Let's study both approaches.***

First approach – Creating auxiliary arrays To better understand this approach, let's use the "from inner to outer" method. When the following code fragment completes its iterations, the auxiliary one-dimensional arrays `minimum` and `maximum` will contain at position 0 the minimum and the maximum values of the first row (row index 0) of array `b` respectively. Assume variable `i` contains value 0.

```
minimum[i] = b[i][0];
maximum[i] = b[i][0];
for (j = 1; j <= COLUMNS - 1; j++) {
   if (b[i][j] < minimum[i]) {
      minimum[i] = b[i][j];
   }
   if (b[i][j] > maximum[i]) {
      maximum[i] = b[i][j];
   }
}
```

✎ *Note that variable* `j` *starts from 1. It wouldn't be wrong to start iterating from column index 0 instead of 1, though the program would perform one useless iteration.*

Now that everything has been clarified, in order to process the whole array `b`, you can just nest the previous code fragment into a for-loop that iterates for all rows as shown next.

```
for (i = 0; i <= ROWS - 1; i++) {
   minimum[i] = b[i][0]; maximum[i] = b[i][0]; for (j = 1; j <= COLUMNS - 1; j++) {
      if (b[i][j] < minimum[i]) {
         minimum[i] = b[i][j];
      }
      if (b[i][j] > maximum[i]) {
         maximum[i] = b[i][j];
      }
   }
}
```

The final Java program is as follows.

```
Project_34.3-7a
 static final int ROWS = 30; static final int COLUMNS =
                            20;
public static void main(String[] args) throws Exception {
                          int i, j;
   double[][] b = new double[ROWS][COLUMNS]; for (i = 0;
                  i <= ROWS - 1; i++) {
            for (j = 0; j <= COLUMNS - 1; j++) {
         b[i][j] = Double.parseDouble(cin.nextLine());
                                }
                              }
   double[] minimum = new double[ROWS]; double[] maximum
   = new double[ROWS]; for (i = 0; i <= ROWS - 1; i++) {
                    minimum[i] = b[i][0];
                    maximum[i] = b[i][0];
            for (j = 1; j <= COLUMNS - 1; j++) {
                  if (b[i][j] < minimum[i]) {
                      minimum[i] = b[i][j];
                                }
                  if (b[i][j] > maximum[i]) {
                      maximum[i] = b[i][j];
                                }
                              }
                            }
            for (i = 0; i <= ROWS - 1; i++) {
      System.out.println(minimum[i] + " " + maximum[i]);
                              }
                            }
```

Second approach – Finding and directly displaying minimum and maximum values Let's use the "from inner to outer" method once again. The next code fragment finds and directly displays the minimum and the maximum values of the first row (row index 0) of array `b`. Assume variable `i` contains the value 0.

```
minimum = b[i][0];
maximum = b[i][0];
for (j = 1; j <= COLUMNS - 1; j++) {
   if (b[i][j] < minimum) {
      minimum = b[i][j];
   }
   if (b[i][j] > maximum) {
      maximum = b[i][j];
   }
}
System.out.println(minimum + " " + maximum);
```

In order to process the whole array `b`, you can just nest this code fragment into a for-loop that iterates for all rows, as follows.

```
for (i = 0; i <= ROWS - 1; i++) {
   minimum = b[i][0]; maximum = b[i][0]; for (j = 1; j <= COLUMNS - 1; j++) {
      if (b[i][j] < minimum) {
         minimum = b[i][j];
      }
      if (b[i][j] > maximum) {
         maximum = b[i][j];
      }
   }
   System.out.println(minimum + " " + maximum); }
```

The final Java program is as follows.

Project_34.3-7b

```
static final int ROWS = 30; static final int COLUMNS =
                          20;
public static void main(String[] args) throws Exception {
            int i, j; double minimum, maximum;
   double[][] b = new double[ROWS][COLUMNS]; for (i = 0;
                  i <= ROWS - 1; i++) {
              for (j = 0; j <= COLUMNS - 1; j++) {
          b[i][j] = Double.parseDouble(cin.nextLine());
                                }
                              }
```

```
            for (i = 0; i <= ROWS - 1; i++) {
                    minimum = b[i][0];
                    maximum = b[i][0];
            for (j = 1; j <= COLUMNS - 1; j++) {
                   if (b[i][j] < minimum) {
                       minimum = b[i][j];
                              }
                   if (b[i][j] > maximum) {
                       maximum = b[i][j];
                              }
                         }
        System.out.println(minimum + " " + maximum);
                        }
                      }
```

34.4 Sorting Arrays

Sorting algorithms are an important topic in computer science. A *sorting algorithm* is an algorithm that puts elements of an array in a certain order. There are many sorting algorithms and each one of them has particular strengths and weaknesses.

Most sorting algorithms work by comparing the elements of the array. They are usually evaluated by their efficiency and their memory requirements.

There are many sorting algorithms. Some of them are: ► the bubble sort algorithm ► the modified bubble sort algorithm ► the selection sort algorithm ► the insertion sort algorithm ► the heap sort algorithm ► the merge sort algorithm ► the quicksort algorithm As regards their efficiency, the bubble sort algorithm is considered the least efficient, while each succeeding algorithm in the list performs better than the preceding one. The quicksort algorithm is considered one of the best and fastest sorting algorithms, especially for large scale data operations.

Sorting algorithms can be used for more than just displaying data in ascending or descending order; they can also assist in finding the minimum and the maximum values from a set of given values. For instance, in an array sorted in ascending order, the minimum value exists at the first index position and the maximum value exists at the last index position. While sorting an array solely for the purpose of finding the minimum and maximum values is very inefficient, if a program sorts an array for other reasons, and you subsequently need the minimum or maximum value, you know where you can find them!

Another scenario where you might need sorting algorithms is when you want to find and display, for example, the three largest (or smallest) numbers in an array. In this case, you can sort the array in descending order and then display only the first three elements, located at index positions 0, 1, and 2.

As you already know, Java incorporates the method `Arrays.sort()` for array sorting. However, there are situations where it's necessary to implement a custom sorting algorithm, especially when you need to sort an array while preserving the one-to-one correspondence with the elements of a second array.

Exercise 34.4-1 The Bubble Sort Algorithm – Sorting One-Dimensional Arrays with Numeric Values

Write a Java program that lets the user enter 20 numerical values into an array and then sorts them in ascending order using the bubble sort algorithm.

Solution

The *bubble sort algorithm* is probably one of the most inefficient sorting algorithms but it is widely used for teaching purposes. The main idea (when asked to sort an array in ascending order) is to repeatedly move the smallest elements of the array to the positions of lowest index. This works as follows: the algorithm iterates through the elements of the array, compares each pair of adjacent elements, and then swaps their contents (if they are in the wrong order). This process is repeated many times until the array is sorted.

For example, let's try to sort the following array in ascending order.

A =

17	*0*
25	*1*
8	*2*
5	*3*
49	*4*
12	*5*

The lowest value is the value 5. According to the bubble sort algorithm, this value must gradually “bubble” or “rise” to position 0, like bubbles rising in a glass of cola. When the value 5 has been moved into position 0, the next smallest value is the value 8. Now, the value 8 must “bubble” to position 1. Next is the value 12, which must “bubble” to position 2, and so on. This process repeats until all elements are placed in proper position.

But how can this “bubbling” be done using an algorithm? Let's see the whole process in more detail. For the previous array A of six elements, five passes must be performed.

First Pass

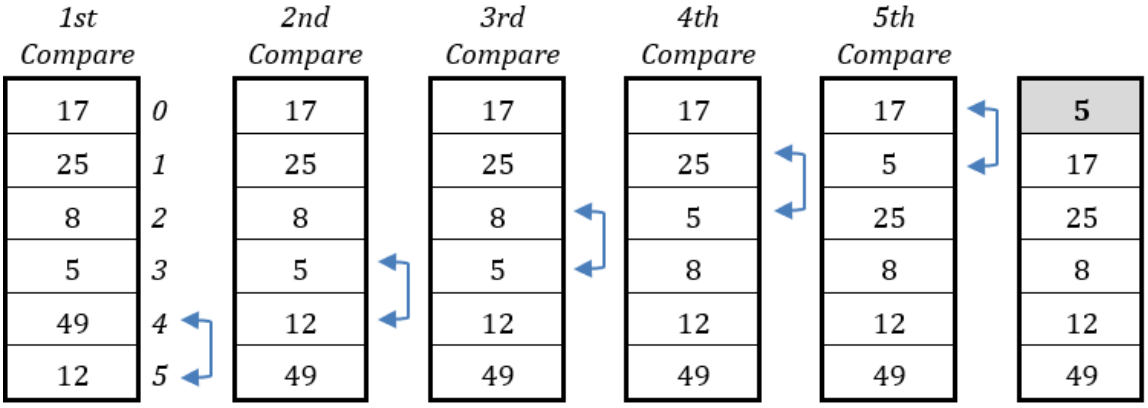
1st Compare Initially, elements at index positions 4 and 5 are compared. Since the value 12 is less than the value 49, these two elements swap their content.

2nd Compare Elements at index positions 3 and 4 are compared. Since the value 12 is **not** less than the value 5, **no** swapping is done.

3rd Compare Elements at index positions 2 and 3 are compared. Since the value 5 is less than the value 8, these two elements swap their content.

4th Compare Elements at index positions 1 and 2 are compared. Since the value 5 is less than the value 25, these two elements swap their content.

5th Compare Elements at index positions 0 and 1 are compared. Since the value 5 is less than the value 17, these two elements swap their content.



The first pass has been completed but, as you can see, the array has not been sorted yet. The only value that is guaranteed to be placed in proper position is the value 5. However, since more passes will follow, there is no need for the value 5 to take part in the subsequent compares. In the pass that follows, one less compare will be performed—that is, four compares.

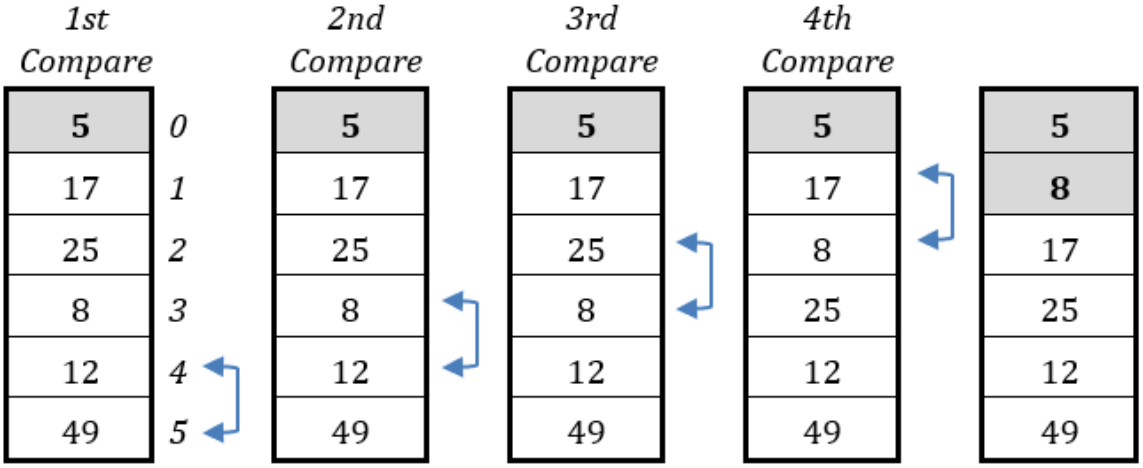
Second Pass

1st Compare Elements at index positions 4 and 5 are compared. Since the value 49 is **not** less than the value 12, **no** swapping is done.

2nd Compare Elements at index positions 3 and 4 are compared. Since the value 12 is **not** less than the value 8, **no** swapping is done.

3rd Compare Elements at index positions 2 and 3 are compared. Since the value 8 is less than the value 25, these two elements swap their content.

4th Compare Elements at index positions 1 and 2 are compared. Since the value 8 is less than the value 17, these two elements swap their content.

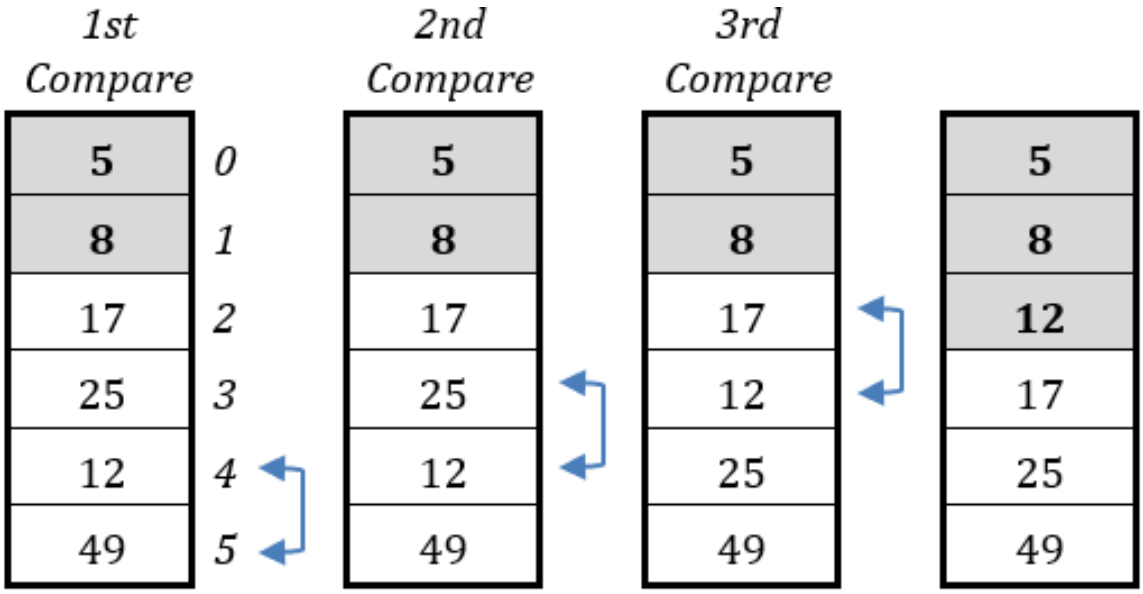


The second pass has been completed and the value of 8 is guaranteed to be placed in proper position. However, since more passes will follow, there is no need for

the value 8 (nor 5, of course) to take part in the subsequent compares. In the pass that follows, one less compare will be performed—that is, three compares.

Third Pass

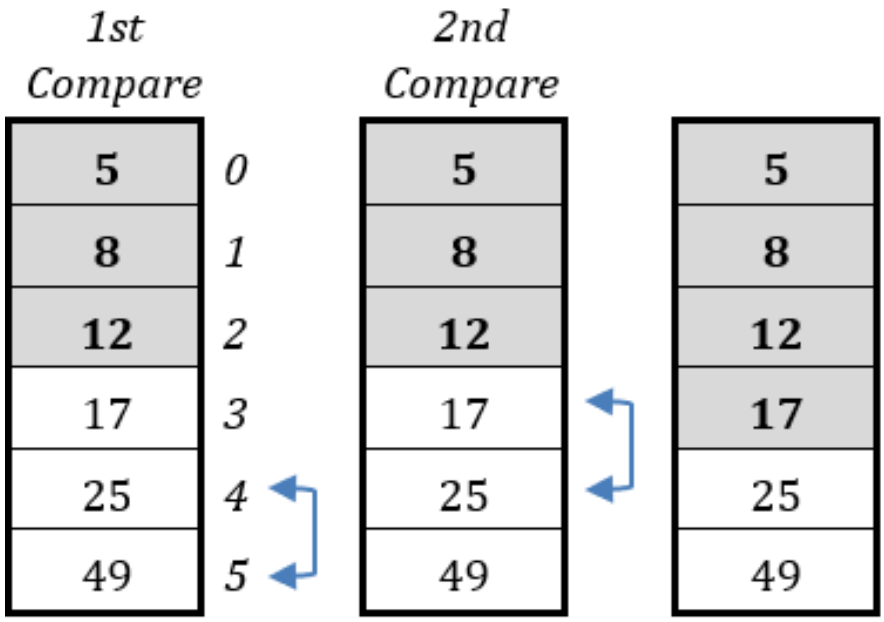
In this pass, three compares (but only two swaps) are performed, as shown below.



The third pass has been completed and the value of 12 is guaranteed to be placed in proper position. As previously, since more passes will follow there is no need for the value 12 (nor the values 5 and 8, of course) to take part in the subsequent compares. In the pass that follows, one compare less will be performed—that is, two compares.

Fourth Pass

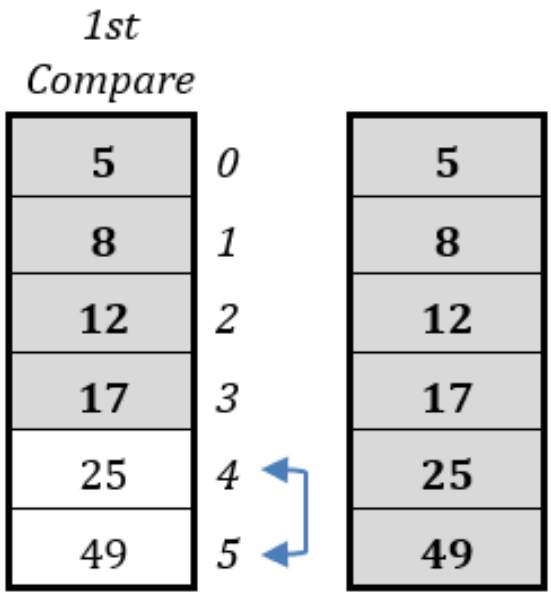
In this pass, two compares (and no swaps) are performed, as shown below.



The fourth pass has been completed and the value 17 is guaranteed to be placed in proper position. As previously, since one last pass will follow, there is no need for the value 17 (nor the values 5, 8, and 12, of course) to take part in the subsequent compares. In the last pass that follows, one compare less will be performed—that is one compare.

Fifth pass

In this last pass, only one compare is performed. Since the value 49 is **not** less than the value 25, **no** swapping is done.



The fifth pass has been completed and the final two values (25 and 49) are now guaranteed to be in proper positions. The bubble sort algorithm has finished and the array is sorted in ascending order!

Now you need a Java program that can do the whole previous process. Let's use the “from inner to outer” method. The code fragment that performs only the first pass is shown below. Please note that this is the inner (nested) loop control structure. Assume variable `m` contains the value 1.

```
for (n = ELEMENTS - 1; n >= m; n--) {
   if (a[n] < a[n - 1]) {
      temp = a[n];
      a[n] = a[n - 1];
      a[n - 1] = temp;
   }
}
```

In the first pass, variable `m` must contain the value 1. This assures that at the last iteration, the elements that are compared are those at index positions 1 and 0.

Swapping the contents of two elements uses a method you have already learned! Please recall the two glasses of orange juice and lemon juice. If this doesn't ring a bell, you need to refresh your memory and re-read Exercise 8.1-3.

The second pass can be performed if you just re-execute the previous code fragment. Variable `m`, however, needs to contain the value 2. This will ensure that the element at index position 0 won't be compared again. Similarly, for the third pass, the previous code fragment can be re-executed but variable `m` needs to contain the value 3 for the same reason.

Accordingly, the previous code fragment needs to be executed five times (one for each pass), and each time variable `m` must be incremented by 1. The final code fragment that sorts array a using the bubble sort algorithm is as follows.

```
for (m = 1; m <= ELEMENTS - 1; m++) {
   for (n = ELEMENTS - 1; n >= m; n--) {
      if (a[n] < a[n - 1]) {
         temp = a[n];
         a[n] = a[n - 1];
         a[n - 1] = temp;
      }
   }
}
```

For N elements, the algorithm needs to perform N − 1 passes. For example, if array `a` contains 20 elements, the statement `for (m = 1; m <= ELEMENTS - 1; m++)` performs 19 passes.

The complete Java program is as follows.

Project_34.4-1

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
   int i, m, n; double temp;
   double[] a = new double[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      a[i] = Double.parseDouble(cin.nextLine());
   }
   for (m = 1; m <= ELEMENTS - 1; m++) {
      for (n = ELEMENTS - 1; n >= m; n--) {
         if (a[n] < a[n - 1]) {
            temp = a[n];
            a[n] = a[n - 1];
            a[n - 1] = temp;
         }
      }
   }
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print(a[i] + "\t");
   }
}
```

✎ *The bubble sort algorithm is very inefficient. The total number of compares that it performs is* $\frac{N(N-1)}{2}$*, where N is the total number of array elements.*

✎ *The total number of swaps depends on the given array. The worst case is when you want to sort in ascending order an array that is already sorted in descending order, or vice versa.*

Exercise 34.4-2 Sorting One-Dimensional Arrays with Alphanumeric Values

Write a code fragment that sorts the alphanumeric values of an array in descending order using the bubble sort algorithm.

Solution

Comparing the wording of this exercise to the previous one, two things are different. First, the bubble sort algorithm needs to sort alphanumeric values, such as names of people or names of cities; and second, it has to sort them in descending order.

In Java you cannot compare strings using comparison operators such as the equal (==), the not equal (!=), the less than (<), or the greater than (>) operators. As you already know from Section 14.3, Java supports two methods: `equals()` and `compareTo()`, which can be used for this purpose. For these methods the letter “A” is considered “less than” the letter “B”, “B” is considered “less than” the letter “C”, and so on. Of course, if the array contains words in which the first letter is identical, Java moves on to compare their second letter and perhaps their

third letter (if necessary). For example, the name “Jonathan” is considered “less than” the name “Jone” as the fourth letter “a” is “less than” the fourth letter “e”. In conclusion, if you replace the Boolean expression `a[n] < a[n - 1]` with the expression `a[n].compareTo(a[n - 1]) < 0`, then the bubble sort algorithm becomes able to sort alphanumeric values in ascending order.

> ✎ *Consider the alphanumeric sorting in the context of how words are organized in an English dictionary.*

Now, let's see what you need to change so that the algorithm can sort in descending order instead of ascending. Do you remember how the bubble sort algorithm actually works? Elements gradually “bubble” to positions of lowest index, like bubbles rise in a glass of cola. What you want in this exercise is to make the bigger (instead of the smaller) elements “bubble” to lower index positions. Therefore, all you need to do is simply reverse the comparison operator of the decision control structure!

The code fragment that sorts alphanumeric values in descending order is as follows.

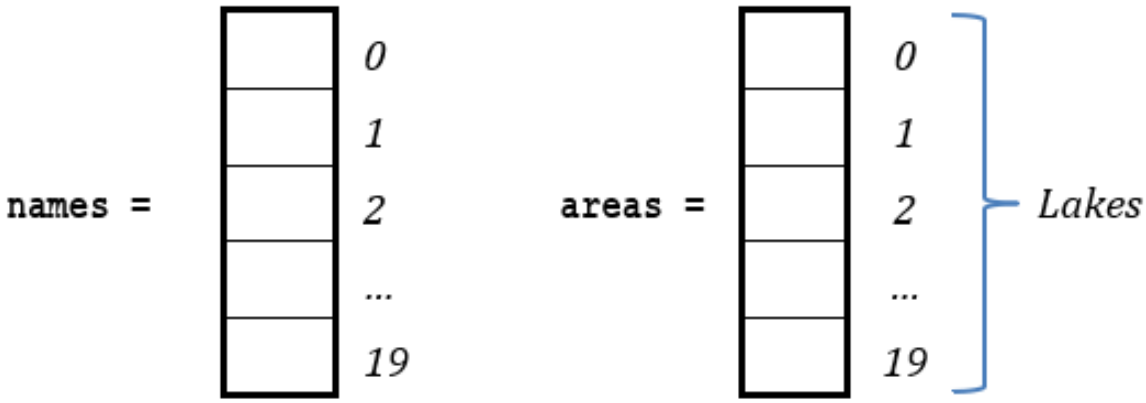
```
String tempStr; for (m = 1; m <= ELEMENTS - 1; m++) {
   for (n = ELEMENTS - 1; n >= m; n--) {
      if (a[n].compareTo(a[n - 1]) > 0) {
         tempStr = a[n];
         a[n] = a[n - 1];
         a[n - 1] = tempStr;
      }
   }
}
```

Exercise 34.4-3 Sorting One-Dimensional Arrays While Preserving the Relationship with a Second Array

Write a Java program that lets the user enter the names of 20 lakes and their corresponding average area. The program must then sort them by average area in ascending order using the bubble sort algorithm.

Solution

In this exercise you need the following two arrays.



If you want to sort array `areas` while preserving the one-to-one correspondence between the elements of the two arrays, you must rearrange the elements of the array `names` as well. This means that every time two elements of the array `areas` swap contents, the corresponding elements of the array `names` must swap contents as well. The Java program is as follows.

Project_34.4-3

```
static final int LAKES = 20;
public static void main(String[] args) throws Exception {
   int i, m, n; double temp; String tempStr;
   String[] names = new String[LAKES]; double[] areas = new double[LAKES]; for (i = 0; i <=
   LAKES - 1; i++) {
      names[i] = cin.nextLine();
      areas[i] = Double.parseDouble(cin.nextLine());
   }
   for (m = 1; m <= LAKES - 1; m++) {
      for (n = LAKES - 1; n >= m; n--) {
         if (areas[n] < areas[n - 1]) {
            temp = areas[n];
            areas[n] = areas[n - 1];
            areas[n - 1] = temp;
            tempStr = names[n];
            names[n] = names[n - 1];
            names[n - 1] = tempStr;
         }
      }
   }
   for (i = 0; i <= LAKES - 1; i++) {
      System.out.println(names[i] + "\t" + areas[i]);
   }
}
```

✎ *Note that you cannot use the variable* `temp` *for swapping the contents of two elements of the array* `names`*; this is because variable* `temp` *is declared as* `double` *while*

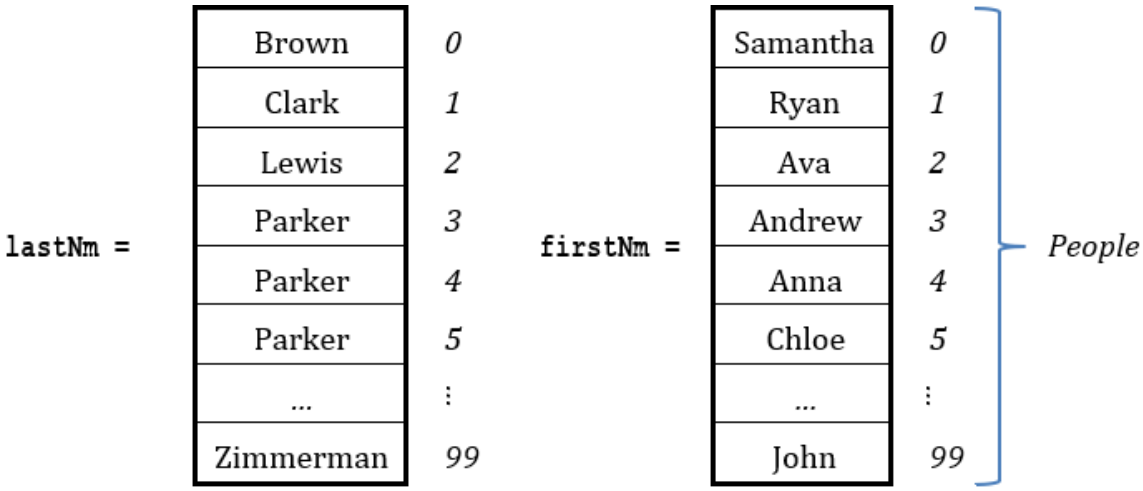
array `names` *contains strings. So, you need a second variable (*`tempStr`*) declared as* `String` *for this purpose.*

Exercise 34.4-4 Sorting Last and First Names

Write a Java program that prompts the user to enter the last and first names of 100 people. The program must then display the names with the last names sorted in alphabetical order. In cases where two or more people share the same last name, their first names should be displayed in alphabetical order.

Solution

You already know how to sort an array while preserving the one-to-one correspondence with the elements of a second array. Now, you have to handle the case when two last names in the first array are equal. According to the wording of the exercise, the corresponding first names in the second array must also be sorted alphabetically. For example, the following array `lastNm` contains the last names of 100 people. It is sorted in alphabetical order and it contains the last name "Parker" three times. The corresponding first names "Andrew", "Anna", and "Chloe", in array `firstNm` also have to be sorted alphabetically, as shown here.



For your convenience, the basic version of the bubble sort algorithm is presented once again here. Please note that this algorithm preserves the one-to-one correspondence between the elements of arrays `lastNm` and `firstNm`.

```
for (m = 1; m <= PEOPLE - 1; m++) {
   for (n = PEOPLE - 1; n >= m; n--) {
      if (lastNm[n].compareTo(lastNm[n - 1]) < 0) {
         tempStr = lastNm[n];
         lastNm[n] = lastNm[n - 1];
         lastNm[n - 1] = tempStr;
         tempStr = firstNm[n];
         firstNm[n] = firstNm[n - 1];
         firstNm[n - 1] = tempStr;
```

```
      }
   }
}
```

✎ *Note that variable* `tempStr` *is used for swapping the contents of the elements of both arrays* `lastNm` *and* `firstNm`*. This is acceptable since both arrays contain strings.*

To solve this exercise, however, this bubble sort algorithm must be adapted correspondingly. According to this basic version of the bubble sort algorithm, when the last name at position `n` is "less" than the last name at position `n - 1`, the algorithm swaps the corresponding contents. However, if the last names at these positions are equal, the algorithm must then verify whether the corresponding first names are in the correct order. If not, a swap is required in the array `firstNm`. The adapted bubble sort algorithm is depicted in the following code fragment.

```
for (m = 1; m <= PEOPLE - 1; m++) {
   for (n = PEOPLE - 1; n >= m; n--) {
      if (lastNm[n].compareTo(lastNm[n - 1]) < 0) {
         tempStr = lastNm[n];
         lastNm[n] = lastNm[n - 1];
         lastNm[n - 1] = tempStr;
         tempStr = firstNm[n];
         firstNm[n] = firstNm[n - 1];
         firstNm[n - 1] = tempStr;
      }
      else if (lastNm[n].equals(lastNm[n - 1])) { //If the last names are equal
         if (firstNm[n].compareTo(firstNm[n - 1]) < 0) { //check the corresponding first
         names
            tempStr = firstNm[n]; //and swap if not in correct order
            firstNm[n] = firstNm[n - 1];
            firstNm[n - 1] = tempStr;
         }
      }
   }
}
```

The final Java program is presented next.

Project_34.4-4

```
static final int PEOPLE = 100;
public static void main(String[] args) throws Exception {
   int i, m, n; String tempStr;
   //Read arrays firstNm and lastNm String[] firstNm = new String[PEOPLE]; String[] lastNm =
   new String[PEOPLE]; for (i = 0; i <= PEOPLE - 1; i++) {
      System.out.print("First name for person No. " + (i + 1) + ": ");
      firstNm[i] = cin.nextLine();
      System.out.print("Last name for person No. " + (i + 1) + ": ");
      lastNm[i] = cin.nextLine();
```

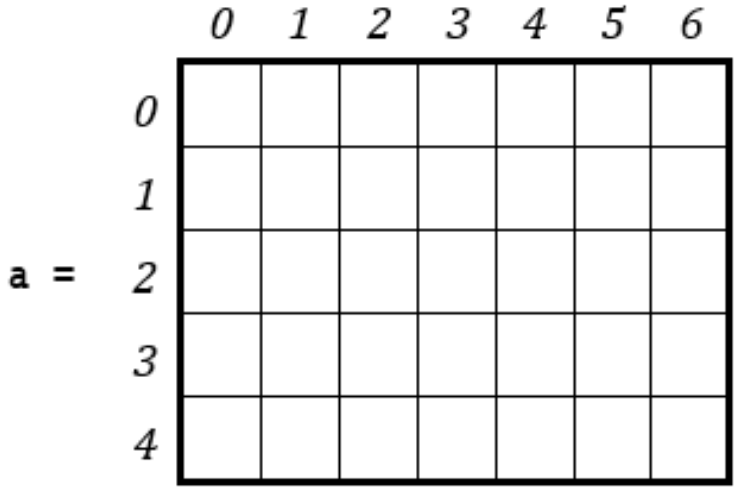
```
   }
   //Sort arrays lastNm and firstNm for (m = 1; m <= PEOPLE - 1; m++) {
      for (n = PEOPLE - 1; n >= m; n--) {
         if (lastNm[n].compareTo(lastNm[n - 1]) < 0) {
            tempStr = lastNm[n];
            lastNm[n] = lastNm[n - 1];
            lastNm[n - 1] = tempStr;
            tempStr = firstNm[n];
            firstNm[n] = firstNm[n - 1];
            firstNm[n - 1] = tempStr;
         }
         else if (lastNm[n].equals(lastNm[n - 1])) {
            if (firstNm[n].compareTo(firstNm[n - 1]) < 0) {
               tempStr = firstNm[n];
               firstNm[n] = firstNm[n - 1];
               firstNm[n - 1] = tempStr;
            }
         }
      }
   }
   //Display arrays lastNm and firstNm for (i = 0; i <= PEOPLE - 1; i++) {
      System.out.println(lastNm[i] + "\t" + firstNm[i]);
   }
}
```

Exercise 34.4-5 Sorting a Two-Dimensional Array

Write a code fragment that sorts each column of a two-dimensional array in ascending order. Assume that the array contains numerical values.

Solution

An example of a two-dimension array is as follows.



Since this array has seven columns, the bubble sort algorithm needs to be executed seven times, one for each column. Therefore, the whole bubble sort

algorithm should be nested within a for-loop that iterates seven times.

But let's get things in the right order. Using the “from inner to outer” method, the next code fragment sorts only the first column (column index 0) of the two-dimensional array `a`. Assume variable `j` contains the value 0.

```
for (m = 1; m <= ROWS - 1; m++) {
   for (n = ROWS - 1; n >= m; n--) {
      if (a[n][j] < a[n - 1][j]) {
         temp = a[n][j];
         a[n][j] = a[n - 1][j];
         a[n - 1][j] = temp;
      }
   }
}
```

Now, in order to sort all columns, you can nest this code fragment in a for-loop that iterates for all of them, as follows.

```
for (j = 0; j <= COLUMNS - 1; j++) {
   for (m = 1; m <= ROWS - 1; m++) {
      for (n = ROWS - 1; n >= m; n--) {
         if (a[n][j] < a[n - 1][j]) {
            temp = a[n][j];
            a[n][j] = a[n - 1][j];
            a[n - 1][j] = temp;
         }
      }
   }
}
```

That wasn't so difficult, was it?

Exercise 34.4-6 The Modified Bubble Sort Algorithm – Sorting One-Dimensional Arrays

Write a Java program that lets the user enter the weights of 20 people and then displays the three heaviest weights and the three lightest weights. Use the ***modified*** *bubble sort algorithm.*

Solution

To solve this exercise, the Java program can sort the user-provided data in ascending order and then display the elements at index positions 17, 18, and 19 (for the three heaviest weights) and the elements at index positions 0, 1 and 2 (for the three lightest weights). But what is that modified version of the bubble sort algorithm, and how does it actually work? Suppose you have the following array containing the weights of six people.

	0	1	2	3	4	5
w =	165	170	180	190	182	200

If you look closer, you can confirm for yourself that the only elements not in the proper position are those at index positions 3 and 4. If you swap their values, the array `w` immediately becomes sorted! Unfortunately, the bubble sort algorithm doesn't operate this way. For this given array of six elements, it will perform five passes either way, with a total of $\frac{N(N-1)}{2} = 15$ compares, where *N* is the total number of array elements. For larger arrays, the total number of compares that the bubble sort algorithm performs increases exponentially! For example, for a given array of 1000 elements, the bubble sort algorithm performs 499,500 compares!

Of course the modified bubble sort algorithm can overcome this situation as follows: if a complete pass is performed and no swaps have been made, then this indicates that the array is now sorted and there is no need for further passes. To accomplish this, the Java program can use a flag variable that indicates if any swaps were made. At the beginning of a pass, a value of `false` can be assigned to the flag variable; when a swap is made, a value of `true` is assigned. If, at the end of the pass, the flag is still `false`, this indicates that no swaps have been made, thus iterations must stop. The modified bubble sort is shown next. It uses the `break` statement and the flag variable `swaps`.

```
for (m = 1; m <= ELEMENTS - 1; m++) {
   //Assign false to variable swaps swaps = false;
   //Perform a new pass for (n = ELEMENTS - 1; n >= m; n--) {
      if (w[n] < w[n - 1]) {
         temp = w[n];
         w[n] = w[n - 1];
         w[n - 1] = temp;
         swaps = true;
      }
   }
   //If variable swaps is still false, no swaps have been made in this pass. Stop iterations!
   if (!swaps) break; }
```

✎ *The value* `false` *must be assigned to variable* `swaps` *each time a new pass starts. This is why the statement* `swaps = false` *must be placed between the two* `for` *statements.*

✎ *The statement* `if (!swaps)` *is equivalent to the statement* `if (swaps == false)`.

The final Java program is shown next.

Project_34.4-6

```
static final int ELEMENTS = 20;
public static void main(String[] args) throws Exception {
   int i, m, n; boolean swaps; double temp;
   double[] w = new double[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      w[i] = Double.parseDouble(cin.nextLine());
   }
   for (m = 1; m <= ELEMENTS - 1; m++) {
      swaps = false;
      for (n = ELEMENTS - 1; n >= m; n--) {
         if (w[n] < w[n - 1]) {
            temp = w[n];
            w[n] = w[n - 1];
            w[n - 1] = temp;
            swaps = true;
         }
      }
      if (!swaps) break;
   }
   System.out.println("The three heaviest weights are:"); System.out.println(w[ELEMENTS - 3]
   + " " + w[ELEMENTS - 2] + " " + w[ELEMENTS - 1]); System.out.println("The three lightest
   weights are:"); System.out.println(w[0] + " " + w[1] + " " + w[2]); }
```

Exercise 34.4-7 The Selection Sort Algorithm – Sorting One-Dimensional Arrays

Write a code fragment that sorts the elements of an array in ascending order using the selection sort algorithm. Assume that the array contains numerical values.

Solution

The *selection sort algorithm* is inefficient for large scale data, as is the bubble sort algorithm, but it generally performs better than the latter. It is the simplest of all the sorting algorithms and performs well on computer systems in which limited main memory (RAM) comes into play.

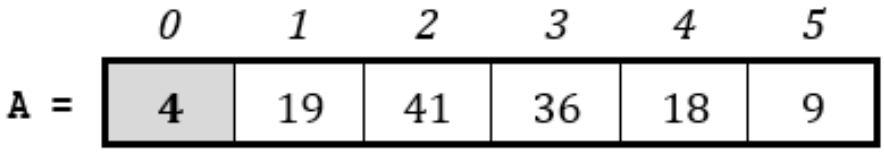
The algorithm finds the smallest (or largest, depending on sorting order) element of the array and swaps its content with that at position 0. Then the process is repeated for the remainder of the array; the next smallest (or largest) element is found and put into the next position, until all elements are examined.

For example, let's try to sort the following array in ascending order.

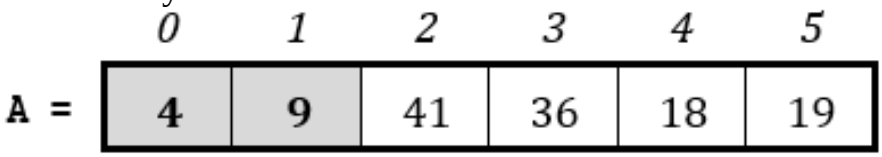
	0	1	2	3	4	5
A =	18	19	39	36	4	9

The lowest value is the value 4, found at position 4. According to the selection sort algorithm, this element swaps its content with the element at position 0. The

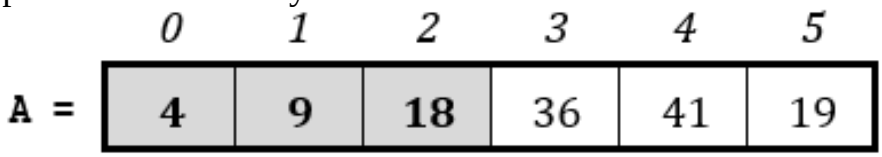
array A becomes



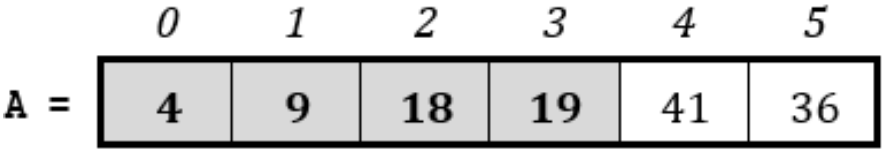
The lowest value in the remainder of the array (index positions 1 to 5) is the value 9, found at position 5. This element swaps its content with the element at position 1. The array A becomes



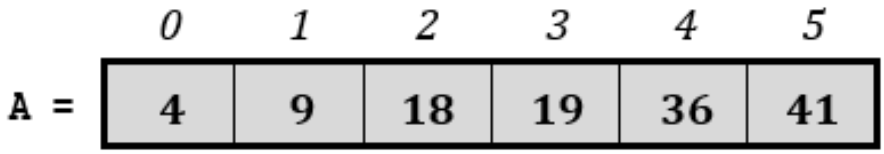
The lowest value in the remainder of the array (index positions 2 to 5) is the value 18, found at position 4. This element swaps its content with the element at position 2. The array A becomes



Proceeding the same way, the next lowest value is the value 19, found at position 5. The array A becomes



The next lowest value is the value 36, found at position 5. This element swaps its content with the element at position 4 and the array A is finally sorted in ascending order!



Now, let's write the corresponding Java program. The “from inner to outer” method is used in order to help you better understand the whole process. The next code fragment finds the smallest element and then swaps its content with that at position 0. Please note that this is the inner (nested) loop control structure. Assume variable m contains the value 0.

```
minimum = a[m];
indexOfMin = m;
for (n = m; n <= ELEMENTS - 1; n++) {
   if (a[n] < minimum) {
      minimum = a[n];
      indexOfMin = n;
```

```
   }
}
//Minimum found! Now, swap values.
temp = a[m];
a[m] = a[indexOfMin];
a[indexOfMin] = temp;
```

Now, in order to repeat the process for all elements of the array, you can nest this code fragment within a for-loop that iterates for all elements. The final selection sort algorithm that sorts an array in ascending order is as follows.

```
for (m = 0; m <= ELEMENTS - 1; m++) {
   minimum = a[m];
   indexOfMin = m;
   for (n = m; n <= ELEMENTS - 1; n++) {
      if (a[n] < minimum) {
         minimum = a[n];
         indexOfMin = n;
      }
   }
   temp = a[m];
   a[m] = a[indexOfMin]; a[indexOfMin] = temp; }
```

If you wish to sort an array in descending order, all you need to do is search for maximum instead of minimum values.

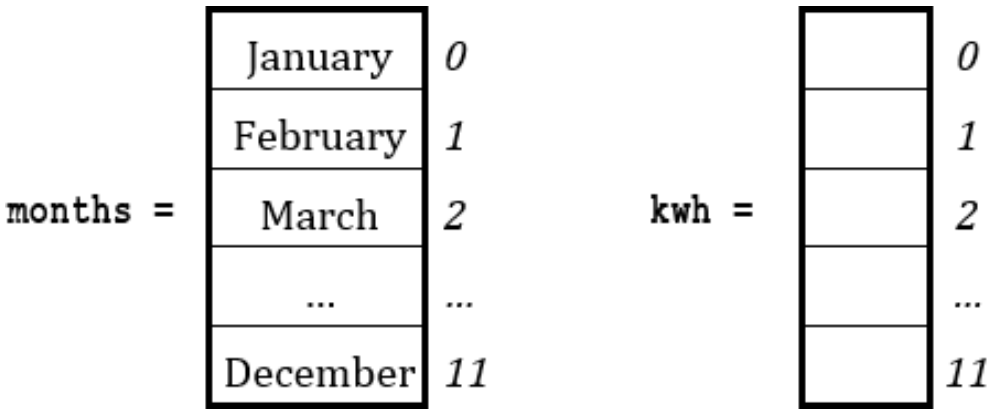
As in the bubble sort algorithm, in order to sort alphanumeric data with the selection sort algorithm, you can simply replace the Boolean expression `a[n] < minimum` *with the expression* `a[n].compareTo(minimum) < 0`.

Exercise 34.4-8 Sorting One-Dimensional Arrays While Preserving the Relationship with a Second Array

Write a Java program that prompts the user to enter the total number of kWh consumed each month for a period of one year. It then displays the three months with the highest consumption of kWh, along with the corresponding number of KWh (in descending order). Use the selection sort algorithm.

Solution

In this exercise you need the following two one-dimensional arrays.



While the selection sort algorithm sorts the elements of array `kwh`, the one-to-one correspondence with the elements of array `months` must be preserved. This means that every time two elements of array `kwh` swap contents, the corresponding elements of array `months` must swap their contents as well.

However, given that you solely require the three months with the highest consumption of KWh, the selection sort algorithm should sort only the first three elements. The Java program is as follows.

Project_34.4-8

```
public static void main(String[] args) throws Exception {
   int i, m, indexOfMax, n; double maximum, temp; String tempStr;
   String[] months = {"January", "February", "March", "April", "May", "June", "July",
   "August", "September", "October", "November", "December"};
   double[] kwh = new double[months.length]; for (i = 0; i <= months.length - 1; i++) {
      System.out.print("Enter kWh for " + months[i] + ": ");
      kwh[i] = Double.parseDouble(cin.nextLine());
   }
   for (m = 0; m <= 2; m++) { //Sort only the first three elements
   maximum = kwh[m];
      indexOfMax = m;
      for (n = m; n <= months.length - 1; n++) {
         if (kwh[n] > maximum) {
            maximum = kwh[n];
            indexOfMax = n;
         }
      }
      //Swap values of kwh
      temp = kwh[m];
      kwh[m] = kwh[indexOfMax];
      kwh[indexOfMax] = temp;
      //Swap values of months
      tempStr = months[m];
      months[m] = months[indexOfMax];
      months[indexOfMax] = tempStr;
   }
```

```
   for (i = 0; i <= 2; i++) {
      System.out.println(months[i] + ": " + kwh[i]);
   }
}
```

✎ *If this exercise required the use of the bubble sort instead of the selection sort algorithm, you could employ the same "trick". The algorithm could perform 3 passes instead of* `ELEMENTS - 1` *passes.*

Exercise 34.4-9 The Insertion Sort Algorithm – Sorting One-Dimensional Arrays

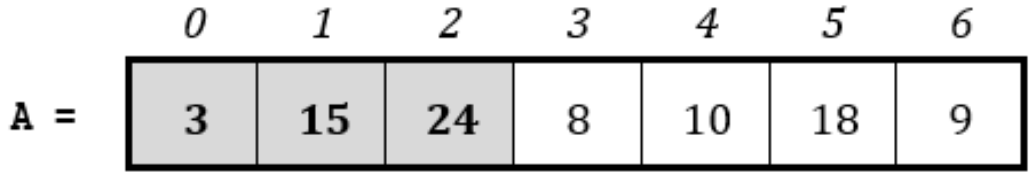
Write a code fragment that sorts the elements of an array in ascending order using the insertion sort algorithm. Assume that the array contains numerical values.

Solution

The *insertion sort algorithm* is inefficient for large scale data, as are the selection and the bubble sort algorithms, but it generally performs better than either of them. Moreover, the insertion sort algorithm can prove very fast when sorting very small arrays— sometimes even faster than the quicksort algorithm.

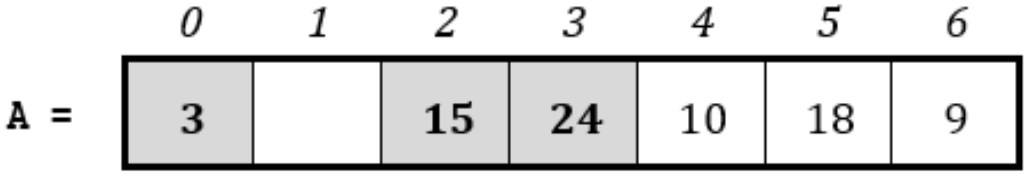
The insertion sort algorithm resembles the way you might sort playing cards. You start with all the cards face down on the table. The cards on the table represent the unsorted "array". In the beginning your left hand is empty, but in the end this hand will hold the sorted cards. The process goes as follows: you remove from the table one card at a time and insert it into the correct position in your left hand. To find the correct position for a card, you compare it with each of the cards already in your hand, from right to left. At the end, there must be no cards on the table and your left hand will hold all the cards, sorted.

For example, let's try to sort the following array in ascending order. To better understand this example, assume that the sorting process has already begun and the first three elements of the array have been sorted.

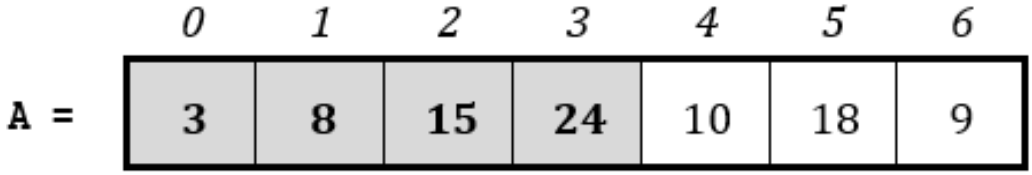


✎ *The elements at index positions 0, 1, and 2 represent the cards in your left hand, while the remaining elements of the array represent the unsorted cards on the table.*

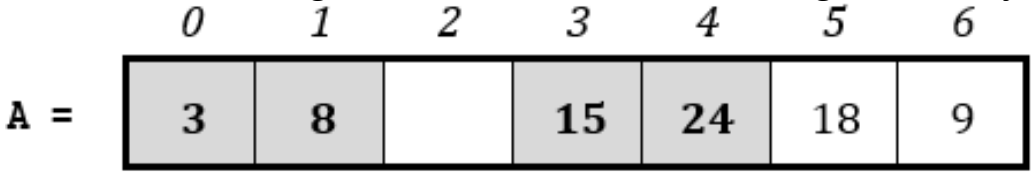
The element at position 3 (which is 8) is removed from the array and all elements on its left with a value greater than 8 are shifted to the right. The array A becomes



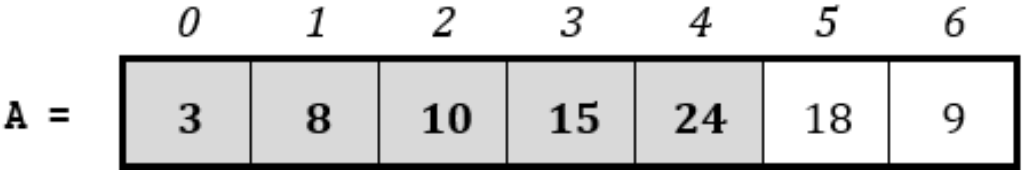
Now that a position has been released, the value 8 is inserted in there. The array becomes



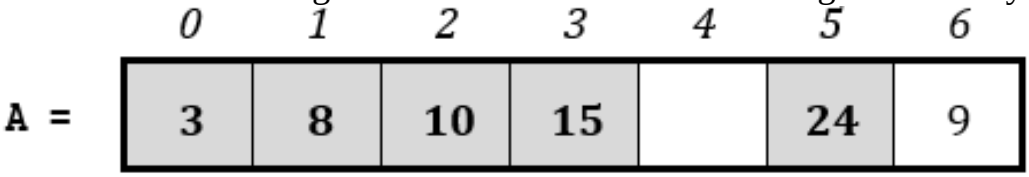
The element at position 4 (which is 10) is removed from the array and all elements on its left with a value greater than 10 are shifted to the right. The array A becomes



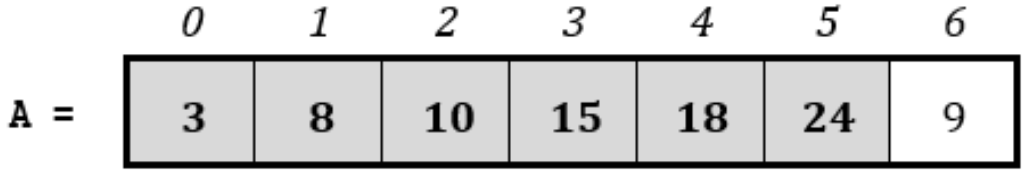
Now that a position has been released, the value of 10 is inserted in there. The array becomes



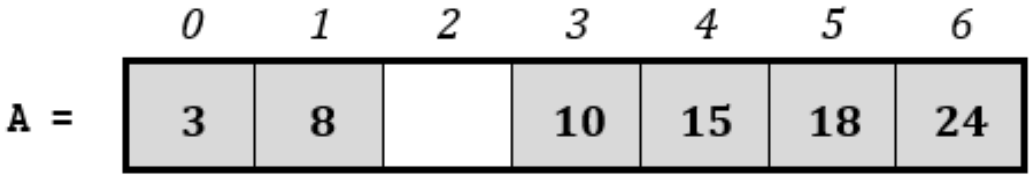
The element at position 5 (which is 18) is removed from the array and all elements on its left with a value greater than 18 are shifted to the right. The array A becomes



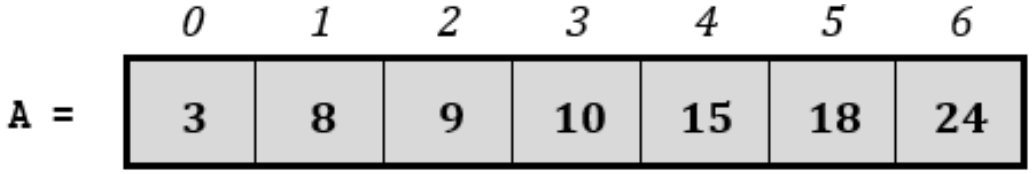
The value of 18 is inserted in the released position. The array becomes



The element at position 6 (which is 9) is removed from the array and all elements on its left with a value greater than 9 are shifted to the right. The array A becomes



Finally, the value of 9 is inserted in the released position, the algorithm finishes and the array is now sorted.



✎ *What the algorithm actually does is to check the unsorted elements one by one and insert each one in the appropriate position among those considered already sorted.*

The code fragment that sorts an array in ascending order using the insertion sort algorithm is as follows.

```
for (m = 1; m <= ELEMENTS - 1; m++) {
   //"Remove" the element at index position m from the array and keep it in variable element
   element = a[m];
   //Shift appropriate elements to the right n = m;
   while (n > 0 && a[n - 1] > element) {
      a[n] = a[n - 1]; //Equivallent to:
      n--; //a[n--] = a[n – 1];
   }
   //Insert the previously "removed" element at index position n a[n] = element;
}
```

✎ *Please note that the element at index position* `m` *is not actually removed from the array but is in fact overwritten when shifting to the right is performed. This is why its value is kept in variable* `element` *before shifting the elements.*

✎ *If you wish to sort an array in descending order, all you need to do is alter the Boolean expression of the* `while` *statement to* `n > 0 && a[n – 1] < element`.

As in the previous two sorting algorithms, in order to sort alphanumeric data in Java, you can simply replace the Boolean expression `n > 0 && a[n - 1] > element` *with the expression* `n > 0 && a[n - 1].compareTo(element) > 0`.

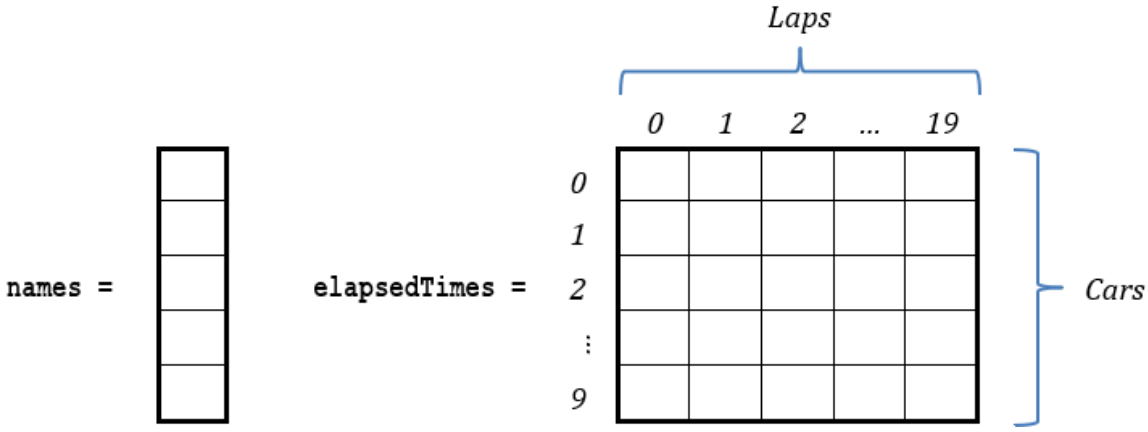
Exercise 34.4-10 The Three Worst Elapsed Times

Ten race car drivers run their cars as fast as possible on a racing track. Each car runs 20 laps and for each lap the corresponding elapsed time (in seconds) is recorded. Write a Java program that prompts the user to enter the name of each

driver and their elapsed time for each lap. The program must then display the name of each driver along with their three worst elapsed times. Use the insertion sort algorithm.

Solution

In this exercise, you need the following two arrays.



After the user enters all data, the Java program must sort each row of the array in descending order but, in the end, must display only the first three columns.

Using the “from inner to outer” method, the next code fragment sorts only the first row (row index 0) of the two-dimensional array `elapsedTimes` in descending order using the insertion sort algorithm. Assume variable `i` contains the value 0.

```
for (m = 1; m <= LAPS - 1; m++) {
   element = elapsedTimes[i][m]; n = m;
   while (n > 0 && elapsedTimes[i][n - 1] < element) {
      elapsedTimes[i][n] = elapsedTimes[i][n - 1];
      n--;
   }
   elapsedTimes[i][n] = element; }
```

Now, in order to sort all rows, you need to nest this code fragment in a for-loop that iterates for all of them, as follows.

```
for (i = 0; i <= CARS - 1; i++) {
   for (m = 1; m <= LAPS - 1; m++) {
      element = elapsedTimes[i][m];
      n = m;
      while (n > 0 && elapsedTimes[i][n - 1] < element) {
         elapsedTimes[i][n] = elapsedTimes[i][n - 1];
         n--;
      }
      elapsedTimes[i][n] = element;
   }
}
```

And now, let's focus on the given exercise. The final Java program is as follows.

Project_34.4-10

```
static final int CARS = 10; static final int LAPS = 20;
public static void main(String[] args) throws Exception {
   int i, j, m, n; double element;
   //Read names and elapsed times all together String[] names = new String[CARS]; double[][]
   elapsedTimes = new double[CARS][LAPS]; for (i = 0; i <= CARS - 1; i++) {
      System.out.print("Enter name for driver No. " + (i + 1) + ": ");
      names[i] = cin.nextLine();
      for (j = 0; j <= LAPS - 1; j++) {
         System.out.print("Enter elapsed time for lap No. " + (j + 1) + ": ");
         elapsedTimes[i][j] = Double.parseDouble(cin.nextLine());
      }
   }
   //Sort array elapsedTimes for (i = 0; i <= CARS - 1; i++) {
      for (m = 1; m <= LAPS - 1; m++) {
         element = elapsedTimes[i][m];
         n = m;
         while (n > 0 && elapsedTimes[i][n - 1] < element) {
            elapsedTimes[i][n] = elapsedTimes[i][n - 1];
            n--;
         }
         elapsedTimes[i][n] = element;
      }
   }
   //Display 3 worst elapsed times for (i = 0; i <= CARS - 1; i++) {
      System.out.println("Worst elapsed times of " + names[i]);
      System.out.println("-----------------------------------");
      for (j = 0; j <= 2; j++) {
         System.out.println(elapsedTimes[i][j]);
      }
   }
}
```

34.5 Searching Elements in Data Structures

In computer science, a *search algorithm* is an algorithm that searches for an item with specific features within a set of data. In the case of a data structure, a search algorithm searches the data structure to find the element, or elements, that equal a given value.

When searching in a data structure, there can be two situations.

- You want to search for a given value in a data structure that may contain the same value multiple times. Therefore, you need to find **all** the elements (or their corresponding indexes) that are equal to that given value.
- You want to search for a given value in a data structure where each value is unique. Therefore, you need to find **just one** element (or its corresponding

index), the one that is equal to that given value, and then stop searching any further!

The most commonly used search algorithms are: ► the linear (or sequential) search algorithm ► the binary search algorithm Both linear and binary search algorithms have advantages and disadvantages.

Exercise 34.5-1 The Linear Search Algorithm – Searching in a One-Dimensional Array that may Contain the Same Value Multiple Times

Write a code fragment that performs a search on a one-dimensional array to find a user-provided value. Assume that the array contains numerical values and may contain the same value multiple times. Use the linear search algorithm.

Solution

The *linear (or sequential) search algorithm* checks if the first element of the array is equal to a given value, then checks the second element, then the third, and so on until the end of the array. Since this process of checking elements one by one is quite slow, the linear search algorithm is suitable for arrays with few elements.

The code fragment is shown next. It looks for a user-provided value `needle` in the array `haystack`!

```
System.out.print("Enter a value to search: "); needle = Double.parseDouble(cin.nextLine());
found = false;
for (i = 0; i <= ELEMENTS - 1; i++) {
   if (haystack[i] == needle) {
      System.out.println(needle + " found at position: " + i);
      found = true;
   }
}
if (!found) {
   System.out.println("Nothing found!"); }
```

Exercise 34.5-2 Display the Last Names of All Those People Who Have the Same First Name

Write a Java program that prompts the user to enter the names of 20 people: their first names into the array `firstNames`*, and their last names into the array* `lastNames`*. The program must then ask the user for a first name, upon which it will search and display the last names of all those whose first name equals the provided one.*

Solution

Even though it is not clear in the wording of the exercise, it is true that the array `firstNames` may contain a value multiple times. How rare is it to meet two people named “John”, for example?

The program must search for the user-provided first name in array `firstNames` and every time it finds it, it must display the corresponding last name from the other array.

The solution is as follows.

Project_34.5-2

```
static final int PEOPLE = 20;
public static void main(String[] args) throws Exception {
   int i; String needle; boolean found;
   String[] firstNames = new String[PEOPLE]; String[] lastNames = new String[PEOPLE]; for (i
   = 0; i <= PEOPLE - 1; i++) {
      System.out.print("Enter first name: ");
      firstNames[i] = cin.nextLine();
      System.out.print("Enter last name: ");
      lastNames[i] = cin.nextLine();
   }
   //Get name to search and convert it to uppercase System.out.print("Enter a first name to
   search: "); needle = cin.nextLine().toUpperCase();
   //Search for user-provided value in array firstNames found = false;
   for (i = 0; i <= PEOPLE - 1; i++) {
      if (firstNames[i].toUpperCase().equals(needle)) { //Convert to uppercase and compare
         System.out.println(lastNames[i]);
         found = true;
      }
   }
   if (!found) {
      System.out.println("No one found!");
   }
}
```

When you want to compare two strings to find out if they are equal, you cannot use the equal (`==`) comparison operator. Java incorporates the special method `equals()` for this purpose.

Since the program deals with alphanumeric data, the `toUpperCase()` method is required so that the program can operate correctly for any user-provided value. For example, if the value "John" exists in the array `firstNames` and the user wants to search for the value "JOHN", the `toUpperCase()` method ensures that the program finds all Johns.

Exercise 34.5-3 The Linear Search Algorithm – Searching in a Two-Dimensional Array that May Contain the Same Value Multiple Times

Write a code fragment that performs a search on each row of a two-dimensional array to find a user-provided value. Assume that the array contains numerical

values and may contain the same value multiple times. Use the linear search algorithm.

Solution

This code fragment must search for the user-provided number in each row of a two-dimensional array that may contain the same value multiple times. This means that the code fragment must search in the first row and display all the columns where the user-provided number is found; otherwise, it must display a message that the user-provided number was not found in the first row. Then, it must search in the second row, and this process must continue until all rows have been examined.

To better understand this exercise, the “inner to outer” method is used. The following code fragment searches for a given value (variable `needle`) only in the first row of the two-dimensional array named `haystack`. Assume variable `i` contains the value 0.

```
found = false;
for (j = 0; j <= COLUMNS - 1; j++) {
   if (haystack[i][j] == needle) {
      System.out.println("Found at column " + j);
      found = true;
   }
}
if (!found) {
   System.out.println("Nothing found in row " + i); }
```

Now, in order to search in all rows, you need to nest this code fragment in a for-loop that iterates for all of them, as follows.

```
System.out.print("Enter a value to search: "); needle = Double.parseDouble(cin.nextLine());
for (i = 0; i <= ROWS - 1; i++) {
   found = false;
   for (j = 0; j <= COLUMNS - 1; j++) {
      if (haystack[i][j] == needle) {
         System.out.println("Found at column " + j);
         found = true;
      }
   }
   if (!found) {
      System.out.println("Nothing found in row " + i);
   }
}
```

Exercise 34.5-4 The Linear Search Algorithm – Searching in a One-Dimensional Array that Contains Unique Values

Write a code fragment that performs a search on a one-dimensional array to find a user-provided value. Assume that the array contains numerical values and each value in the array is unique. Use the linear search algorithm.

Solution

This case is quite different from the previous ones. Since each value in the array is unique, when the user-provided value is found, there is no need to iterate without reason until the end of the array, thus wasting CPU time. There are three approaches, actually! Let's analyze them all!

First approach – Using the `break` statement In this approach, when the user-provided value is found, a `break` statement is used to break out of the for-loop. The solution is as follows.

```
System.out.print("Enter a value to search: "); needle = Double.parseDouble(cin.nextLine());
found = false;
for (i = 0; i <= ELEMENTS - 1; i++) {
   if (haystack[i] == needle) {
      System.out.println(needle + " found at position: " + i);
      found = true;
      break;
   }
}
if (!found) {
   System.out.println("Nothing found!"); }
```

Or you can do the same, in a little bit different way.

```
System.out.print("Enter a value to search: "); needle = Double.parseDouble(cin.nextLine());
indexPosition = -1;
for (i = 0; i <= ELEMENTS - 1; i++) {
   if (haystack[i] == needle) {
      indexPosition = i;
      break;
   }
}
if (indexPosition == -1) {
   System.out.println("Nothing found!"); }
else {
   System.out.println(needle + " found at position: " + indexPosition); }
```

Second approach – Using a flag The `break` statement doesn't actually exist in all computer languages; and since this book's intent is to teach you "Algorithmic Thinking" (and not just special statements that only Java supports), let's look at an alternate approach.

In the next code fragment, when the user-provided value is found within array `haystack`, the variable `found` forces the flow of execution to immediately exit the loop.

```
System.out.print("Enter a value to search: "); needle = Double.parseDouble(cin.nextLine());
found = false;
i = 0;
while (i <= ELEMENTS - 1 && !found) {
   if (haystack[i] == needle) {
      found = true;
      indexPosition = i;
   }
   i++;
}
if (!found) {
   System.out.println("Nothing found!"); }
else {
   System.out.println(needle + " found at position: " + indexPosition); }
```

Third approach – Using only a pre-test loop structure This approach is likely the most efficient among the three. The while-loop iterates through the array, comparing each element with the `needle`. The loop continues as long as two conditions are met: variable `i` (representing the index) is within the valid range for the array `haystack`, and the value at the current index in the array is not equal to the `needle`. If both conditions are `true`, variable `i` is incremented to move to the next element. This process continues until a match is found or until the end of the array is reached.

```
System.out.print("Enter a value to search: "); needle = Double.parseDouble(cin.nextLine());
i = 0;
while (i < ELEMENTS - 1 && haystack[i] != needle) {
   i++;
}
if (haystack[i] != needle) {
   System.out.println("Nothing found!"); }
else {
   System.out.println(needle + " found at position: " + i); }
```

Exercise 34.5-5 Searching for a Social Security Number

In the United States, the Social Security Number (SSN) is a nine-digit identity number applied to all U.S. citizens in order to identify them for the purposes of Social Security. Write a Java program that prompts the user to enter the SSN and the first and last names of 100 people. The program must then ask the user for an SSN, upon which it will search and display the first and last name of the person who holds that SSN.

Solution

In the United States, there is no possibility that two or more people will have the same SSN. Thus, even though it is not clear in the wording of the exercise, each value in the array that holds the SSNs is unique!

According to everything you have learned so far, the solution to this exercise is as follows.

Project_34.5-5

```
static final int PEOPLE = 100;
public static void main(String[] args) throws Exception {
   int i; String needle;
   String[] SSNs = new String[PEOPLE]; String[] firstNames = new String[PEOPLE]; String[]
   lastNames = new String[PEOPLE]; for (i = 0; i <= PEOPLE - 1; i++) {
      System.out.print("Enter SSN: ");
      SSNs[i] = cin.nextLine();
      System.out.print("Enter first name: ");
      firstNames[i] = cin.nextLine();
      System.out.print("Enter last name: ");
      lastNames[i] = cin.nextLine();
   }
   System.out.print("Enter an SSN to search: "); needle = cin.nextLine();
   //Search for user-provided value in array SSNs i = 0;
   while (i < PEOPLE - 1 && !SSNs[i].equals(needle)) {
      i++;
   }
   if (!SSNs[i].equals(needle)) {
      System.out.println("Nothing found!");
   }
   else {
      System.out.println(firstNames[i] + " " + lastNames[i]);
   }
}
```

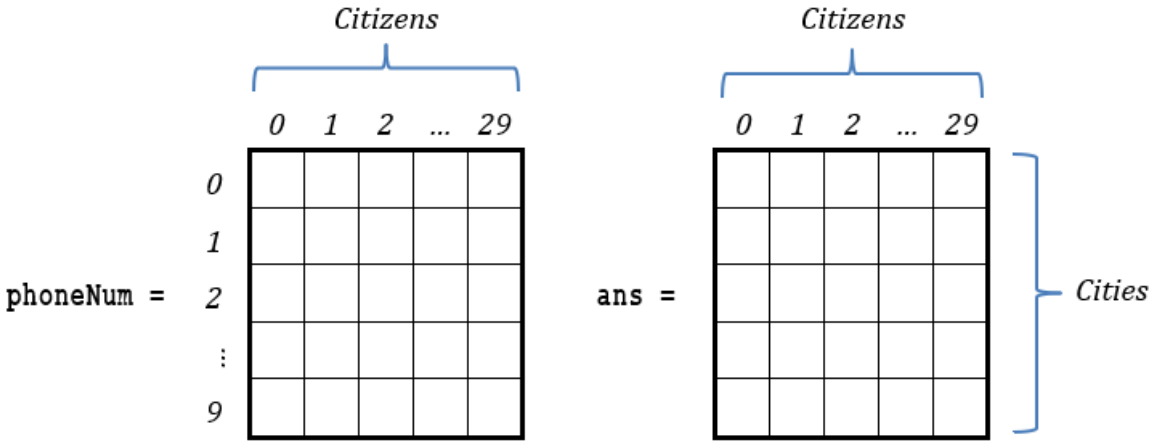
When you want to compare two strings to find out if they are not equal, you cannot use the not equal (`!=`) comparison operator. Java incorporates the special method `equals()` for this purpose. When this method returns `false`, it means that the two strings are not equal.

Exercise 34.5-6 The Linear Search Algorithm – Searching in a Two-Dimensional Array that Contains Unique Values

A public opinion polling company makes phone calls in 10 cities and asks 30 citizens in each city whether or not they exercise. Write a Java program that prompts the user to enter each citizen's phone number and their answer (Y for Yes, N for No, S for Sometimes). The program must then prompt the user to enter a phone number, and it will search and display the answer that was provided at this phone number. The program must also validate data input and accept only the values Y, N, or S as an answer.

Solution

In this exercise, you need the following two arrays.



Even though it is not clear in the wording of the exercise, each value in the array phoneNum is unique! The program must search for the user-provided number and if it finds it, it must stop searching thereafter. The solution is as follows.

Project_34.5-6

```
static final int CITIES = 10; static final int CITIZENS = 30;
public static void main(String[] args) throws Exception {
   int i, j, positionI, positionJ; boolean found; String needle;
   String[][] phoneNum = new String[CITIES][CITIZENS]; String[][] ans = new String[CITIES]
   [CITIZENS]; for (i = 0; i <= CITIES - 1; i++) {
      System.out.println("City No. " + (i + 1));
      for (j = 0; j <= CITIZENS - 1; j++) {
         System.out.print("Enter phone number of citizen No. " + (j + 1) + ": ");
         phoneNum[i][j] = cin.nextLine();
         System.out.print("Enter the answer of citizen No. " + (j + 1) + ": ");
         ans[i][j] = cin.nextLine().toUpperCase();
         while (!ans[i][j].equals("Y") && !ans[i][j].equals("N") && !ans[i][j].equals("S")) {
            System.out.print("Wrong answer. Enter a valid one: ");
            ans[i][j] = cin.nextLine().toUpperCase();
         }
      }
   }
   System.out.print("Enter a phone number to search: "); needle = cin.nextLine();
   found = false;
   positionI = -1;
   positionJ = -1;
   for (i = 0; i <= CITIES - 1; i++) {
      for (j = 0; j <= CITIZENS - 1; j++) {
         if (phoneNum[i][j].equals(needle)) { //If it is found
            found = true;
            positionI = i; //Keep row index where needle was found
            positionJ = j; //Keep column index where needle was found
            break; //Exit the inner loop
         }
```

```
      }
      if (found)
         break; //If it is found, exit the outer loop as well
   }
   if (!found) {
      System.out.println("Phone number not found!");
   }
   else {
      System.out.print("Phone number " + phoneNum[positionI][positionJ] + " gave '");
      switch (ans[positionI][positionJ]) {
         case "Y":
            System.out.print("Yes");
            break;
         case "N":
            System.out.print("No");
            break;
         default:
            System.out.print("Sometimes");
      }
      System.out.println("' as an answer");
   }
}
```

Exercise 34.5-7 Checking if a Value Exists in all Columns

Write a Java program that lets the user enter numeric values into a 20 × 30 array. After all of the values have been entered, the program then lets the user enter a value. In the end, a message must be displayed if the user-provided value exists, at least once, in each column of the array.

Solution

This exercise can be solved using the linear search algorithm and a counter variable `count`. The Java program will iterate through the first column; if the user-provided value is found, the Java program must stop searching in the first column thereafter, and the variable `count` must increment by one. Then, the program will iterate through the second column; if the user-provided value is found again, the Java program must stop searching in the second column thereafter, and the variable `count` must once more increment by one. This process must repeat until all columns have been examined. At the end of the process, if the value of `count` is equal to the total number of columns, this means that the user-provided value exists, at least once, in each column of the array.

Let's use the “from inner to outer” method. The following code fragment searches in first column (column index 0) of the array and if the user-provided value is found, the flow of execution exits the for-loop and variable `count` increments by one. Assume variable `j` contains the value 0.

```
found = false;
for (i = 0; i <= ROWS - 1; i++) {
   if (haystack[i][j] == needle) {
      found = true;
      break;
   }
}
if (found) {
   count++;
}
```

Now you can nest this code fragment in a for-loop that iterates for all columns.

```
for (j = 0; j <= COLUMNS - 1; j++) {
   found = false;
   for (i = 0; i <= ROWS - 1; i++) {
      if (haystack[i][j] == needle) {
         found = true;
         break;
      }
   }
   if (found) {
      count++;
   }
}
```

You are almost ready—but consider a small detail! If the inner for-loop doesn't find the user-provided value in a column, the outer for-loop must stop iterating. It is pointless to continue because the user-provided value does not exist in at least one column. Thus, a better approach would be to use a `break` statement for the outer loop as shown in the code fragment that follows.

```
for (j = 0; j <= COLUMNS - 1; j++) {
   found = false;
   for (i = 0; i <= ROWS - 1; i++) {
      if (haystack[i][j] == needle) {
         found = true;
         break;
      }
   }
   if (found) {
      count++;
   }
   else {
      break;
   }
}
```

The final Java program is as follows.

Project_34.5-7

```
static final int ROWS = 20; static final int COLUMNS = 30;
public static void main(String[] args) throws Exception {
   int i, j, count; boolean found; double needle;
   double[][] haystack = new double[ROWS][COLUMNS]; for (i = 0; i <= ROWS - 1; i++) {
      for (j = 0; j <= COLUMNS - 1; j++) {
         haystack[i][j] = Double.parseDouble(cin.nextLine());
      }
   }
   System.out.print("Enter a value to search: "); needle =
   Double.parseDouble(cin.nextLine());
   count = 0;
   for (j = 0; j <= COLUMNS - 1; j++) {
      found = false;
      for (i = 0; i <= ROWS - 1; i++) {
         if (haystack[i][j] == needle) {
            found = true;
            break;
         }
      }
      if (found) {
         count++;
      }
      else {
         break;
      }
   }
   if (count == COLUMNS) {
      System.out.println(needle + " found in every column!");
   }
}
```

✎ *If you need a message to be displayed when a user-provided value exists at least once in each **row** (rather than in each column), the Java program can follow a procedure like the one previously shown but in this case, it must iterate through the rows instead of the columns.*

Exercise 34.5-8 The Binary Search Algorithm – Searching in a Sorted One-Dimensional Array

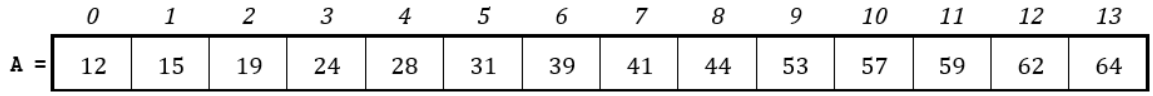
Write a code fragment that performs a search on a sorted one-dimensional array to find a given value. Use the binary search algorithm.

Solution

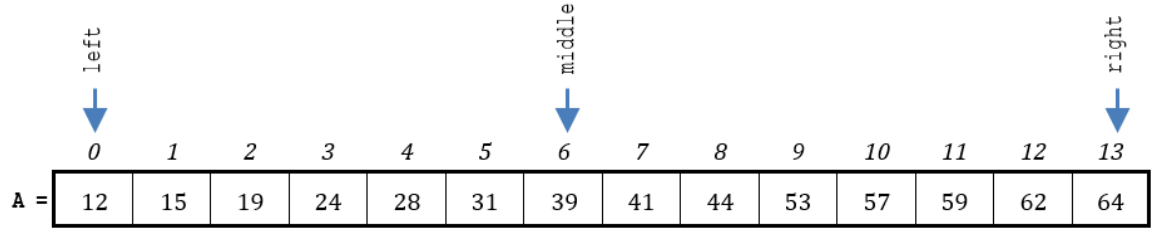
The *binary search algorithm* is considered very fast and can be used with large scale data. Its main disadvantage, though, is that the data need to be sorted.

The main idea of the binary search algorithm is to first examine the element in the middle of the array. If it does not match the “needle in the haystack” that you are looking for, the algorithm determines whether the target is smaller or larger than the middle element. This guides the search to the corresponding half of the array. In other words, if the “needle” you are looking for is smaller than the value of the middle element, it means that the “needle” might be in the first half of the array; otherwise it might be in the last half of the array. The process continues, narrowing down the search by checking the middle element in the remaining half of the array until the "needle" is found or the portion of the array being examined is reduced to a single element. If the latter occurs without finding the "needle," it means the “needle” is not present in the array.

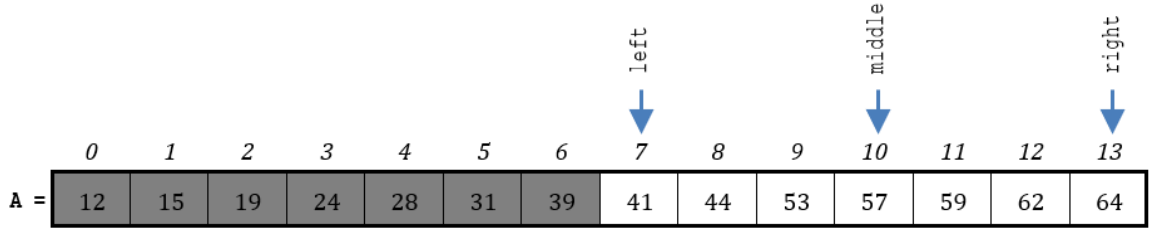
Confused? Let's try to analyze the binary search algorithm through an example. The following array contains numeric values in ascending order. Assume that the “needle” that you are looking for is the value 44.



Three variables are used. Initially, variable `left` contains the value 0 (this is the index of the first element), variable `right` contains the value 13 (this is the index of the last element) and variable `middle` contains the value 6 (this is approximately the index of the middle element).

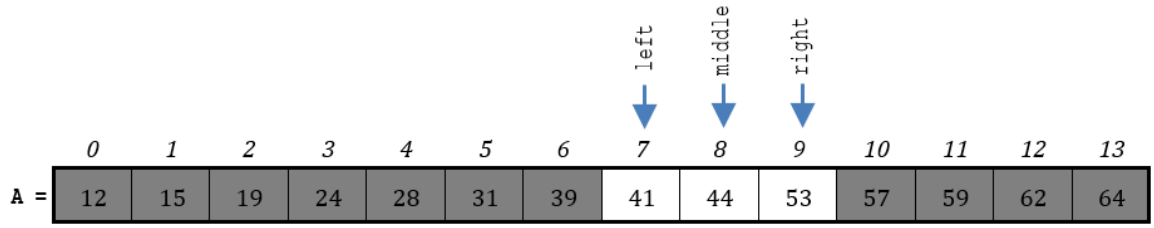


The “needle” (value 44) that you are looking for is larger than the value of 39 in the middle, thus the element that you are looking for might be in the last half of the array. Therefore, variable `left` is updated to point to index position 7 and variable `middle` is updated to a point in the middle between `left` (the new one) and `right`, as shown below.



Now, the “needle” (value 44) that you are looking for is smaller than the value of 57 in the middle, thus the element that you are looking for might be in the first half of the portion of the array being examined. Therefore, it is the variable `right`

that is now updated to point to index position 9, and variable `middle` is updated to point to the middle between `left` and `right` (the new one), as shown below.



You are done! The “needle” has been found at index position 8 and the whole process can stop!

Each unsuccessful comparison reduces the number of elements left to check by half!

The index variables `left` *and* `right` *each time point to the beginning and end, respectively, of the portion of the array being examined.*

Now, let's see the corresponding code fragment.

```
left = 0;
right = ELEMENTS - 1;
found = false;
while (left <= right && !found) {
   middle = (int)((left + right) / 2); //This is a DIV 2 operation
   if (needle < haystack[middle]) { //If the needle is in the first half of the portion
   right = middle - 1; //of the array being examined, update the right index
   }
   else if (needle > haystack[middle]) { //If it is in the second half,
   left = middle + 1; //update the left index
   }
   else {
      found = true;
   }
}
if (!found) {
   System.out.println("Nothing found!"); }
else {
   System.out.println(needle + " found at position: " + middle); }
```

The binary search algorithm is very efficient because it drastically reduces the search space with each iteration, making it highly effective for sorted arrays. Using this algorithm on the example array, the value of 44 can be found within just three iterations. In contrast, the linear search algorithm would require nine iterations for the same data!

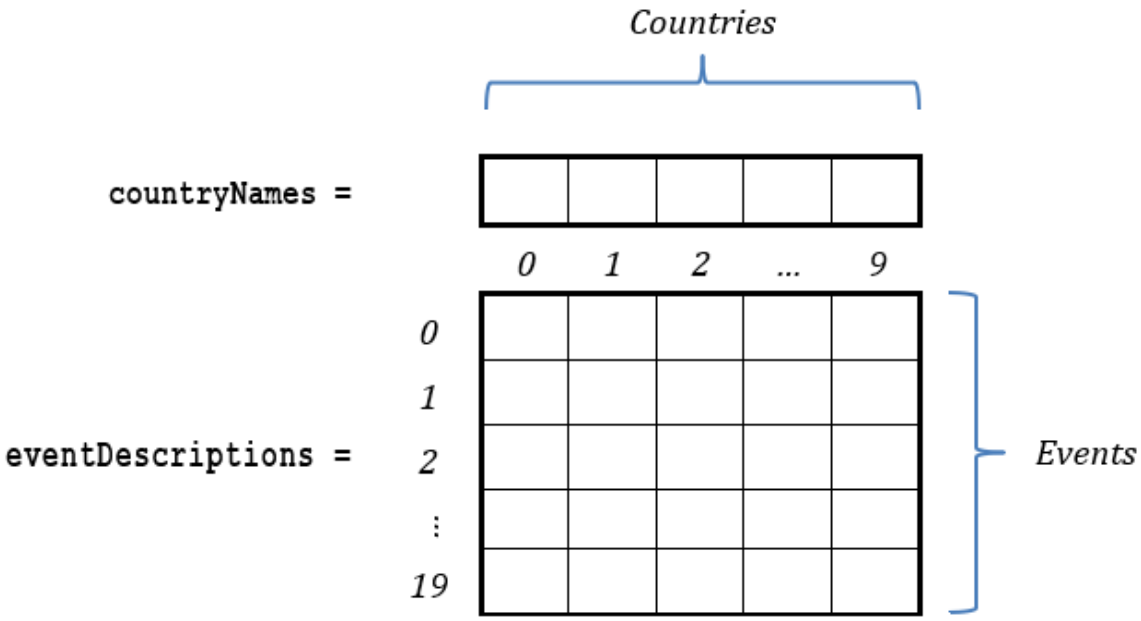
If the array contains a value multiple times, the binary search algorithm can find only one occurrence.

Exercise 34.5-9 Display all the Historical Events for a Country

Write a Java program that prompts the user to enter the names of 10 countries in alphabetical order and 20 important historical events for each country (a brief description of each event). The Java program must then prompt the user to enter a country, and it will search and display all events for that country. Use the binary search algorithm.

Solution

In this exercise, the following two arrays are required.



Assume that the user enters a country to search for, and the binary search algorithm finds that country, for example, at index position 2 of array `countryNames`. The program can then use this value of 2 as a column index for the array `eventDescriptions`, and display all the event descriptions of column 2.

The Java program is as follows.

Project_34.5-9

```
static final int EVENTS = 20; static final int COUNTRIES = 10;
public static void main(String[] args) throws Exception {
   int j, i, left, right, middle; boolean found; String needle;
   String[] countryNames = new String[COUNTRIES]; String[][] eventDescriptions = new
   String[EVENTS][COUNTRIES]; for (j = 0; j <= COUNTRIES - 1; j++) {
      System.out.print("Enter Country No. " + (j + 1) + ": ");
      countryNames[j] = cin.nextLine();
      for (i = 0; i <= EVENTS - 1; i++) {
         System.out.print("Enter description for event No. " + (i + 1) + ": ");
         eventDescriptions[i][j] = cin.nextLine();
```

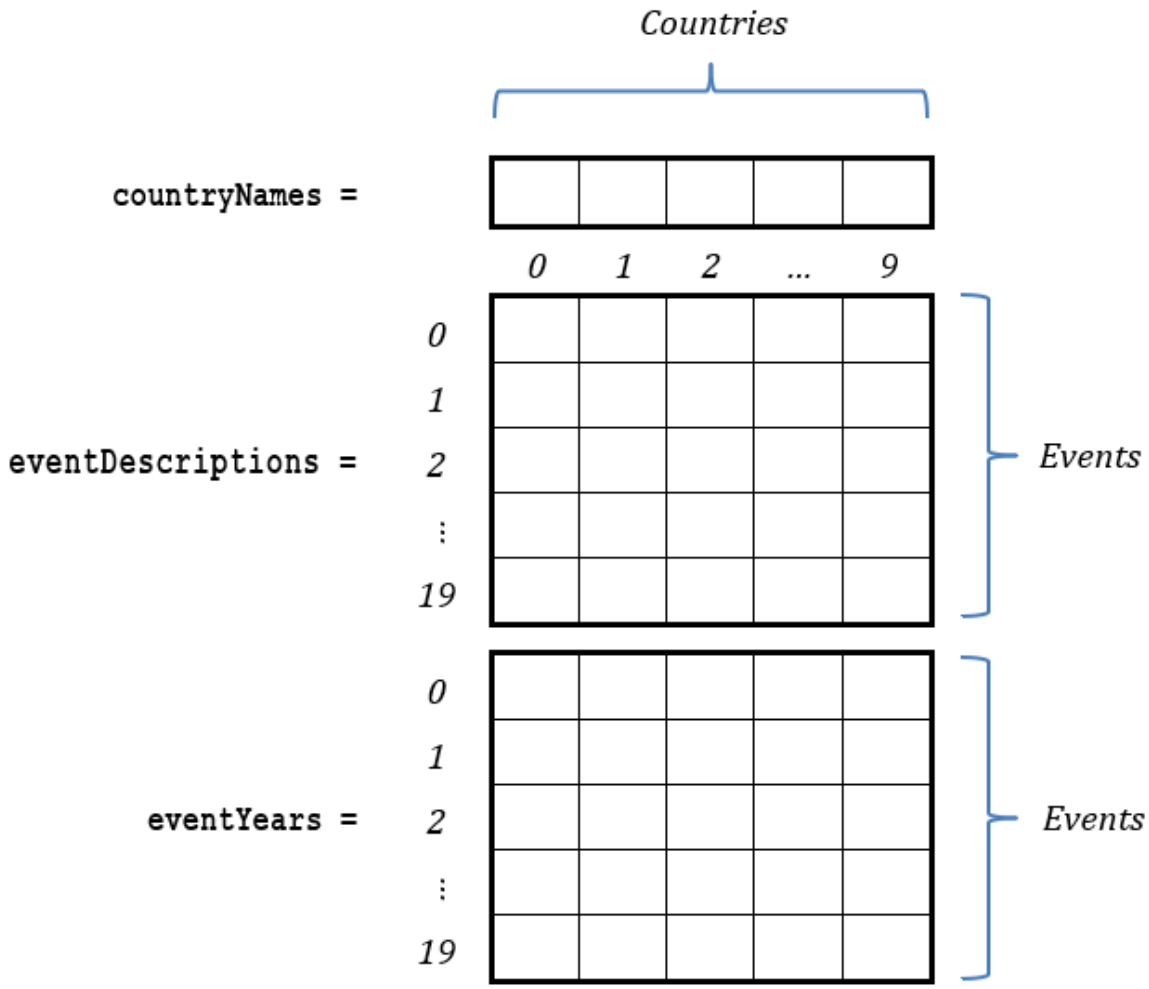
```
      }
   }
   System.out.print("Enter a country to search: "); needle = cin.nextLine().toUpperCase();
   //Country names are entered in alphabetical order.
   //Use the binary search algorithm to search for needle.
   middle = -1;
   left = 0;
   right = EVENTS - 1;
   found = false;
   while (left <= right && !found) {
      middle = (int)((left + right) / 2);
      if (needle.compareTo(countryNames[middle].toUpperCase()) < 0) {
         right = middle - 1;
      }
      else if (needle.compareTo(countryNames[middle].toUpperCase()) > 0) {
         left = middle + 1;
      }
      else {
         found = true;
      }
   }
   if (!found) {
      System.out.println("No country found!");
   }
   else {
      for (i = 0; i <= EVENTS - 1; i++) {
         System.out.println(eventDescriptions[i][middle]);
      }
   }
}
```

Exercise 34.5-10 Searching in Each Column of a Two-Dimensional Array

Write a Java program that prompts the user to enter the names of 10 countries and 20 important historical events for each country (a brief description of each event), and the corresponding year of each event. The Java program must then prompt the user to enter a year, and it will search and display all events that happened that year for each country. Use the binary search algorithm. Assume that for each country there is only one event in each year and that the user enters the events ordered by year in ascending order.

Solution

In this exercise, the following three arrays are required.



In order to write the code fragment that performs a search on each column of the array `eventYears`, let's use the “from inner to outer” method. The next binary search algorithm searches in the first column (column index 0) for a user-provided year. Assume variable `j` contains the value 0. Since the search is performed vertically, and in order to increase program's readability, the variables `left` and `right` of the binary search algorithm have been renamed to `top` and `bottom` respectively.

```
top = 0;
bottom = EVENTS - 1;
found = false;
while (top <= bottom && !found) {
   middle = (int)((top + bottom) / 2);
   if (needle < eventYears[middle][j]) {
      bottom = middle - 1;
   }
   else if (needle > eventYears[middle][j]) {
      top = middle + 1;
   }
```

```
   else {
      found = true;
   }
}
if (!found) {
   System.out.println("No event found for country " + countryNames[j]); }
else {
   System.out.println("Country: " + countryNames[j]); System.out.println("Year: " +
   eventYears[middle][j]); System.out.println("Event: " + eventDescriptions[middle][j]); }
```

Now, nesting this code fragment in a for-loop that iterates for all columns results in the following.

```
for (j = 0; j <= COUNTRIES - 1; j++) {
   top = 0;
   bottom = EVENTS - 1;
   found = false;
   while (top <= bottom && !found) {
      middle = (int)((top + bottom) / 2);
      if (needle < eventYears[middle][j]) {
         bottom = middle - 1;
      }
      else if (needle > eventYears[middle][j]) {
         top = middle + 1;
      }
      else {
         found = true;
      }
   }
   if (!found) {
      System.out.println("No event found for country " + countryNames[j]);
   }
   else {
      System.out.println("Country: " + countryNames[j]);
      System.out.println("Year: " + eventYears[middle][j]);
      System.out.println("Event: " + eventDescriptions[middle][j]);
   }
}
```

The final Java program is as follows.

Project_34.5-10

```
static final int EVENTS = 20; static final int COUNTRIES = 10;
public static void main(String[] args) throws Exception {
   int j, i, needle, top, bottom, middle; boolean found;
   String[] countryNames = new String[COUNTRIES]; String[][] eventDescriptions = new
   String[EVENTS][COUNTRIES]; int[][] eventYears = new int[EVENTS][COUNTRIES]; for (j = 0; j
   <= COUNTRIES - 1; j++) {
      System.out.print("Enter Country No. " + (j + 1) + ": ");
      countryNames[j] = cin.nextLine();
```

```
      for (i = 0; i <= EVENTS - 1; i++) {
         System.out.print("Enter description for event No. " + (i + 1) + ": ");
         eventDescriptions[i][j] = cin.nextLine();
         System.out.print("Enter year for event No. " + (i + 1) + ": ");
         eventYears[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   System.out.print("Enter a year to search: "); needle = Integer.parseInt(cin.nextLine());
   middle = -1;
   for (j = 0; j <= COUNTRIES - 1; j++) {
      top = 0;
      bottom = EVENTS - 1;
      found = false;
      while (top <= bottom && !found) {
         middle = (int)((top + bottom) / 2);
         if (needle < eventYears[middle][j]) {
            bottom = middle - 1;
         }
         else if (needle > eventYears[middle][j]) {
            top = middle + 1;
         }
         else {
            found = true;
         }
      }
      if (!found) {
         System.out.println("No event found for country " + countryNames[j]);
      }
      else {
         System.out.println("Country: " + countryNames[j]);
         System.out.println("Year: " + eventYears[middle][j]);
         System.out.println("Event: " + eventDescriptions[middle][j]);
      }
   }
}
```

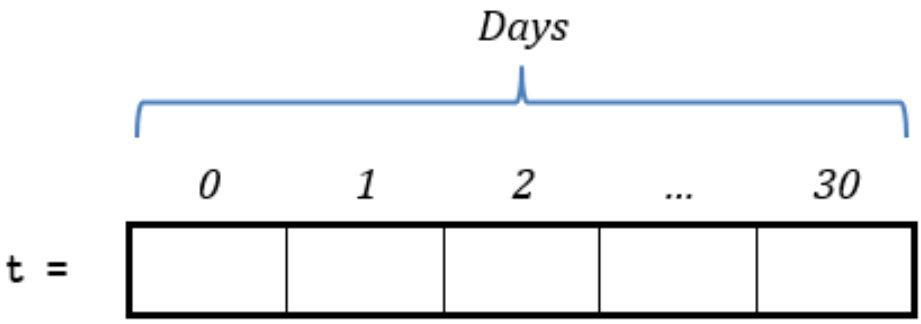
34.6 Exercises of a General Nature with Data Structures

Exercise 34.6-1 On Which Days was There a Possibility of Snow?

Write a Java program that lets the user enter the temperatures (in degrees Fahrenheit) recorded at the same hour each day for the 31 days of January. The Java program must then display the numbers of those days (1, 2, ..., 31) on which there was a possibility of snow, that is, those on which temperatures were below 36 degrees Fahrenheit (about 2 degrees Celsius).

Solution

The one-dimensional array for this exercise is shown next.



and the Java program is as follows.

Project_34.6-1

```
static final int DAYS = 31;
public static void main(String[] args) throws Exception {
   int i;
   int[] t = new int[DAYS]; for (i = 0; i <= DAYS - 1; i++) {
      t[i] = Integer.parseInt(cin.nextLine());
   }
   for (i = 0; i <= DAYS - 1; i++) {
      if (t[i] < 36) {
         System.out.println((i + 1) + "\t");
      }
   }
}
```

Exercise 34.6-2 Was There Any Possibility of Snow?

Write a Java program that lets the user enter the temperatures (in degrees Fahrenheit) recorded at the same hour each day for the 31 days of January. The Java program must then display a message indicating if there was a possibility of snow, that is, if there were any temperatures below 36 degrees Fahrenheit (about 2 degrees Celsius).

Solution

In this exercise, you **cannot** replicate the approach used in the previous exercise. The code fragment that follows would be **incorrect.**

```
for (i = 0; i <= DAYS - 1; i++) {
   if (t[i] < 36) {
      System.out.println("There was a possibility of snow in January!");
   }
}
```

If January had more than one day with a temperature below 36 degrees Fahrenheit, the same message would be displayed multiple times—and obviously you do not want this! You actually want to display a message once, regardless of whether January had one, two, or even more days below 36 degrees Fahrenheit.

There are two approaches, actually. Let's study them both.

First approach – Counting all temperatures below 36 degrees Fahrenheit In this approach, you can use a variable in the program to count all the days on which the temperature was below 36 degrees Fahrenheit. After all of the days have been examined, the program can check the value of this variable. If the value is not zero, it means that there was at least one day where there was a possibility of snow.

```
Project_34.6-2a
static final int DAYS = 31;
public static void main(String[] args) throws Exception {
int i, count;
int[] t = new int[DAYS]; for (i = 0; i <= DAYS - 1; i++) {
t[i] = Integer.parseInt(cin.nextLine());
}
count = 0;
for (i = 0; i <= DAYS - 1; i++) {
if (t[i] < 36) {
count++;
}
}
if (count != 0) {
System.out.println("There was a possibility of snow in
January!");
}
}
```

Second approach – Using a flag In this approach, instead of counting all those days that had a temperature below 36 degrees Fahrenheit, you can use a Boolean variable (a flag). The solution is presented next.

```
Project_34.6-2b
static final int DAYS = 31;
public static void main(String[] args) throws Exception {
int i; boolean found;
int[] t = new int[DAYS]; for (i = 0; i <= DAYS - 1; i++) {
t[i] = Integer.parseInt(cin.nextLine());
}
found = false;
for (i = 0; i <= DAYS - 1; i++) {
if (t[i] < 36) {
found = true;
```

```
                    break;
                  }
                }
            if (found) {
    System.out.println("There was a possibility of snow in
                        January!");
                }
              }
```

✎ *Imagine the variable* `found` *as if it's a real flag. Initially, the flag is not hoisted* (`found = false`). *Within the for-loop, however, when a temperature below 36 degrees Fahrenheit is found, the flag is hoisted* (*the value* `true` *is assigned to the variable* `found`) *and it is never lowered again.*

✎ *Note the* `break` *statement! Once a temperature below 36 degrees Fahrenheit is found, it is meaningless to continue checking thereafter.*

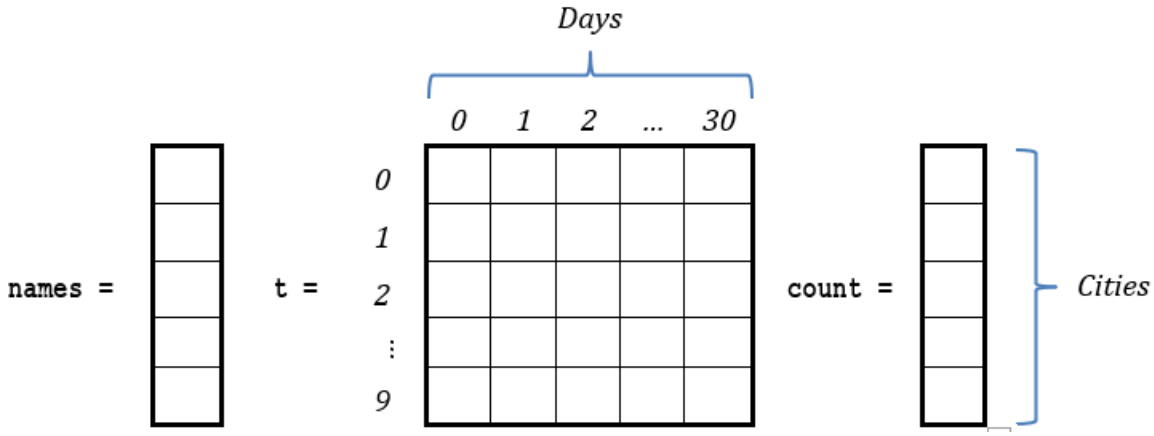
✎ *If the loop performs all of its iterations and no temperature below 36 degrees Fahrenheit is found, the variable* `found` *will still contain its initial value* (`false`) *since the flow of execution never entered the decision control structure.*

Exercise 34.6-3 In Which Cities was There a Possibility of Snow?

Write a Java program that prompts the user to enter the names of ten cities and their temperatures (*in degrees Fahrenheit*) *recorded at the same hour each day for the 31 days of January. The Java program must display the names of the cities in which there was a possibility of snow, that is, those in which temperatures were below 36 degrees Fahrenheit* (*about 2 degrees Celsius*).

Solution

As in the previous exercise, you need to display each city name once, regardless of whether it had one, two, or even more days below 36 degrees Fahrenheit. There are two approaches. In the first approach, the auxiliary array `count`, as presented below, is created by the program to count the total number of days on which each city had temperatures lower than 36 degrees Fahrenheit. The second approach, however, doesn't create the auxiliary array `count`. It uses just one extra Boolean variable (a flag). Obviously the second one is more efficient. But let's study both approaches.



First approach – Using an auxiliary array You were taught in Section 33.2 how to process each row individually. The nested loop control structure that can create the auxiliary array `count` is as follows.

```
int[] count = new int[CITIES]; for (i = 0; i <= CITIES - 1; i++) {
   count[i] = 0;
   for (j = 0; j <= DAYS - 1; j++) {
      if (t[i][j] < 36) {
         count[i]++;
      }
   }
}
```

After array `count` is created you can iterate through it, and when an element contains a value other than zero, it means that the corresponding city had at least one day below 36 degrees Fahrenheit; thus the program must display the name of that city. The final Java program is presented next Project_34.6-3a

```
static final int CITIES = 10; static final int DAYS = 31;
public static void main(String[] args) throws Exception {
    int i, j;
    String[] names = new String[CITIES]; int[][] t = new int[CITIES][DAYS];
    for (i = 0; i <= CITIES - 1; i++) {
        System.out.print("Enter a name for city No: " + (i + 1) + ": ");
        names[i] = cin.nextLine();
        for (j = 0; j <= DAYS - 1; j++) {
            System.out.print("Enter a temperature for day No: " + (j + 1) + ": ");
            t[i][j] = Integer.parseInt(cin.nextLine());
        }
    }
    //Create auxiliary array count int[] count = new int[CITIES]; for (i = 0; i <=
    CITIES - 1; i++) {
        count[i] = 0;
        for (j = 0; j <= DAYS - 1; j++) {
```

```
            if (t[i][j] < 36) {
                count[i]++;
            }
        }
    }
    System.out.println("Cities in which there was a possibility of snow in
    January: "); for (i = 0; i <= CITIES - 1; i++) {
        if (count[i] != 0) {
            System.out.println(names[i]);
        }
    }
}
```

Second approach – Using a flag This approach does not use an auxiliary array. It processes array `t` and directly displays any city name that had a temperature below 36 degrees Fahrenheit. But how can this be done without displaying a city name twice, or even more than twice? This is where you need a flag, that is, an extra Boolean variable.

To better understand this approach, let's use the “from inner to outer” method. The following code fragment checks if the first row of array `t` (row index 0) contains at least one temperature below 36 degrees Fahrenheit; if so, it displays the corresponding city name that exists at position 0 of the array `names`. Assume variable `i` contains the value 0.

```
found = false;
for (j = 0; j <= DAYS - 1; j++) {
   if (t[i][j] < 36) {
      found = true;
      break;
   }
}
if (found) {
   System.out.println(names[i]); }
```

Now that everything has been clarified, in order to process the whole array `t`, you can just nest this code fragment in a for-loop that iterates for all cities, as follows.

```
for (i = 0; i <= CITIES - 1; i++) {
   found = false;
   for (j = 0; j <= DAYS - 1; j++) {
      if (t[i][j] < 36) {
         found = true;
         break;
      }
   }
   if (found) {
```

```
        System.out.println(names[i]);
    }
}
```

The final Java program is as follows.

```
                    Project_34.6-3b
static final int CITIES = 10; static final int DAYS = 31;
public static void main(String[] args) throws Exception {
                    int i, j; boolean found;
 int[] names = new int[CITIES]; int[][] t = new int[CITIES]
        [DAYS]; for (i = 0; i <= CITIES - 1; i++) {
    System.out.print("Enter a name for city No: " + (i + 1)
                             + ": ");
          names[i] = Integer.parseInt(cin.nextLine());
                for (j = 0; j <= DAYS - 1; j++) {
       System.out.print("Enter a temperature for day No: "
                          + (j + 1) + ": ");
           t[i][j] = Integer.parseInt(cin.nextLine());
                                }
                               }
      System.out.println("Cities in which there was a
    possibility of snow in January: "); for (i = 0; i <=
                     CITIES - 1; i++) {
                         found = false;
                for (j = 0; j <= DAYS - 1; j++) {
                        if (t[i][j] < 36) {
                            found = true;
                               break;
                                }
                               }
                          if (found) {
                    System.out.println(names[i]);
                                }
                               }
                               }
```

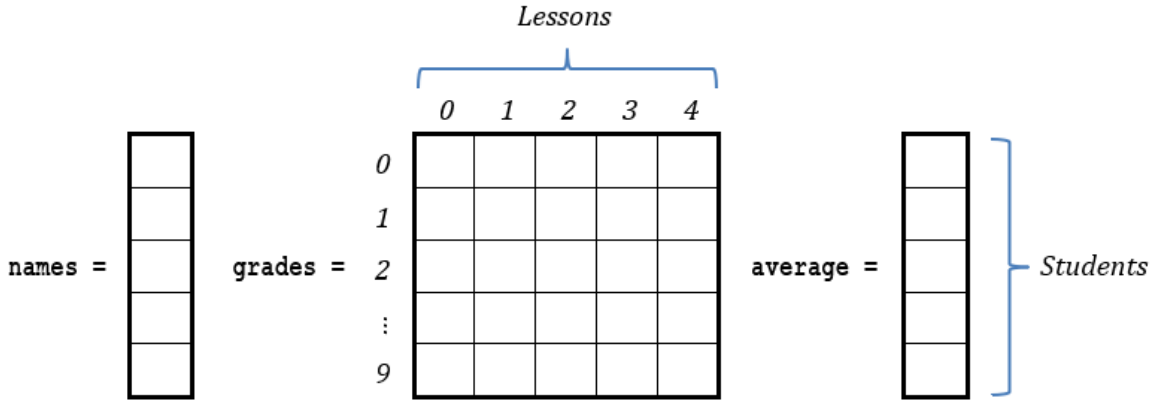
Exercise 34.6-4 Display from Highest to Lowest Grades by Student, and in Alphabetical Order

There are 10 students and each one of them has received their grades for five lessons. Write a Java program that prompts a teacher to enter the name of each student and their grades for all lessons. The program must then calculate each

student's average grade, and display the names and the average grades of the students sorted by their average grade in descending order. Moreover, if two or more students have the same average grade, their names must be displayed in alphabetical order. Use the bubble sort algorithm, adapted accordingly.

Solution

In this exercise, you need the following three arrays. The values for the arrays `names` and `grades` will be entered by the user, whereas the auxiliary array `average` will be created by the Java program.



You're already familiar with all the steps in this exercise. You can create the auxiliary array `average` (see Section 33.2), sort it while maintaining the one-to-one correspondence with the elements in the array `names` (as shown in Exercise 34.4-3), and handle the scenario where, if two average grades are equal, the corresponding student names should be sorted alphabetically (as demonstrated in Exercise 34.4-4). Here's the final Java program.

Project_34.6-4

```
static final int STUDENTS = 10; static final int LESSONS = 5;
public static void main(String[] args) throws Exception {
   int i, j, m, n; double temp; String tempStr;
   //Read array names and grades String[] names = new String[STUDENTS]; int[][] grades = new
   int[STUDENTS][LESSONS]; for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.print("Enter name for student No. " + (i + 1) + ": ");
      names[i] = cin.nextLine();
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("Enter grade for lesson No. " + (j + 1) + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Create array average double[] average = new double[STUDENTS]; for (i = 0; i <= STUDENTS
   - 1; i++) {
      average[i] = 0;
      for (j = 0; j <= LESSONS - 1; j++) {
         average[i] += grades[i][j];
```

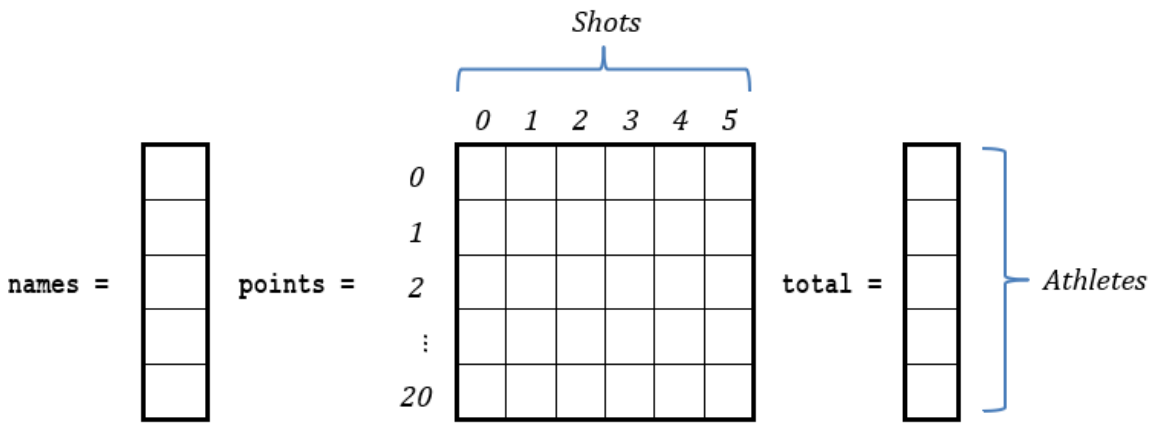
```
      }
      average[i] /= LESSONS;
   }
   //Sort arrays average and names for (m = 1; m <= STUDENTS - 1; m++) {
      for (n = STUDENTS - 1; n >= m; n--) {
         if (average[n] > average[n - 1]) {
            temp = average[n];
            average[n] = average[n - 1];
            average[n - 1] = temp;
            tempStr = names[n];
            names[n] = names[n - 1];
            names[n - 1] = tempStr;
         }
         else if (average[n] == average[n - 1]) {
            if (names[n].compareTo(names[n - 1]) < 0) {
               tempStr = names[n];
               names[n] = names[n - 1];
               names[n - 1] = tempStr;
            }
         }
      }
   }
   //Display arrays names and average for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.println(names[i] + "\t" + average[i]);
   }
}
```

Exercise 34.6-5 Archery at the Summer Olympics

In archery at the Summer Olympics, 20 athletes each shoot six arrows. Write a Java program that prompts the user to enter the name of each athlete, and the points awarded for each shot. The program must then display the names of the three athletes that won the gold, silver, and bronze medals depending on which athlete obtained the highest sum of points. Assume that no two athletes have an equal sum of points.

Solution

In this exercise, you need the following three arrays. The values for the arrays `names` and `points` will be entered by the user, whereas the auxiliary array `total` will be created by the Java program.



After the auxiliary array `total` is created, a sorting algorithm can sort the array `total` in descending order (while preserving the one-to-one correspondence with the elements of the array `names`). The Java program can then display the names of the three athletes at index positions 0, 1, and 2 (since these are the athletes that should win the gold, the silver, and the bronze medals, respectively).

The following program uses the bubble sort algorithm to sort the array `total`. Since the algorithm must sort in descending order, bigger elements must gradually "bubble" to positions of lowest index, like bubbles rise in a glass of cola. However, instead of performing 19 passes (there are 20 athletes), given that only the three best athletes must be found, the algorithm can perform just 3 passes. Doing this, only the first three bigger elements will gradually "bubble" to the first three positions in the array.

The solution is presented next.

Project_34.6-5

```
static final int ATHLETES = 20; static final int SHOTS = 6;
public static void main(String[] args) throws Exception {
   int i, j, m, n, temp; String tempStr;
   //Read array names and points String[] names = new String[ATHLETES]; int[][] points = new
   int[ATHLETES][SHOTS]; for (i = 0; i <= ATHLETES - 1; i++) {
      System.out.print("Enter name for athlete No. " + (i + 1) + ": ");
      names[i] = cin.nextLine();
      for (j = 0; j <= SHOTS - 1; j++) {
         System.out.print("Enter points for shot No. " + (j + 1) + ": ");
         points[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Create array total int[] total = new int[ATHLETES]; for (i = 0; i <= ATHLETES - 1; i++)
   {
      total[i] = 0;
      for (j = 0; j <= SHOTS - 1; j++) {
         total[i] += points[i][j];
      }
```

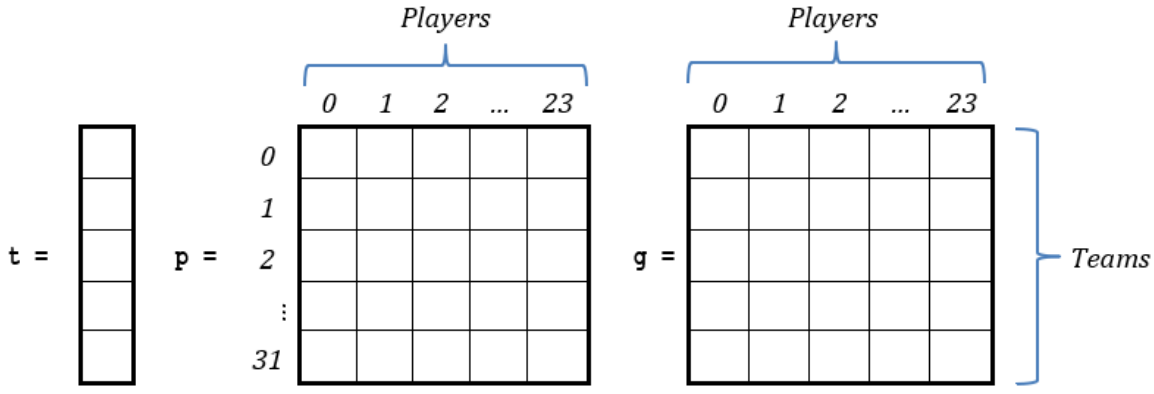
```
   }
   //Sort arrays names and total. Perform only 3 passes for (m = 1; m <= 3; m++) {
      for (n = ATHLETES - 1; n >= m; n--) {
         if (total[n] > total[n - 1]) {
            temp = total[n];
            total[n] = total[n - 1];
            total[n - 1] = temp;
            tempStr = names[n];
            names[n] = names[n - 1];
            names[n - 1] = tempStr;
         }
      }
   }
   //Display gold, silver and bronze metal for (i = 0; i <= 2; i++) {
      System.out.println(names[i] + "\t" + total[i]);
   }
}
```

Exercise 34.6-6 The Five Best Scorers

Write a Java program that prompts the user to enter the names of the 32 national teams of the FIFA World Cup, the names of the 24 players for each team, and the total number of goals each player scored. The program must then display the name of each team along with its five best scorers. Use the bubble sort algorithm.

Solution

In this exercise you need the following three arrays.



✎ *To save paper short array names are used, but it is more or less obvious that array* `t` *holds the names of the 32 national teams, array* `p` *holds the names of the 24 players of each team, and array* `g` *holds the total number of goals each player scored.*

The Java program must sort each row of array `g` in descending order but it must also take care to preserve the one-to-one correspondence with the elements of array `p`. This means that, every time the bubble sort algorithm swaps the contents

of two elements of array g, the corresponding elements of array p must be swapped as well. However, instead of performing 23 passes (there are 24 players), given that only the five best scorers must be found, the algorithm can perform just 5 passes. When sorting is completed, the five best scorers should appear in the first five columns.

The “from inner to outer” method is used again. The following code fragment sorts the first row (row index 0) of array g in descending order and, at the same time, takes care to preserve the one-to-one correspondence with the elements of array p. Assume variable i contains the value 0.

```
for (m = 1; m <= 5; m++) { //Perform 5 passes for (n = PLAYERS - 1; n >= m; n--) {
      if (g[i][n] < g[i][n - 1]) {
         temp = g[i][n];
         g[i][n] = g[i][n - 1];
         g[i][n - 1] = temp;
         tempStr = p[i][n];
         p[i][n] = p[i][n - 1];
         p[i][n - 1] = tempStr;
      }
   }
}
```

Now, in order to sort all rows, you need to nest this code fragment in a for-loop that iterates for all of them, as shown next.

```
for (i = 0; i <= TEAMS - 1; i++) {
   for (m = 1; m <= 5; m++) { //Perform 5 passes
   for (n = PLAYERS - 1; n >= m; n--) {
         if (g[i][n] < g[i][n - 1]) {
            temp = g[i][n];
            g[i][n] = g[i][n - 1];
            g[i][n - 1] = temp;
            tempStr = p[i][n];
            p[i][n] = p[i][n - 1];
            p[i][n - 1] = tempStr;
         }
      }
   }
}
```

The final Java program is as follows.

Project_34.6-6

```
static final int TEAMS = 32; static final int PLAYERS = 24;
public static void main(String[] args) throws Exception {
   int i, j, m, n, temp; String tempStr;
   //Read team names, player names and goals all together String[] t = new String[TEAMS];
   String[][] p = new String[TEAMS][PLAYERS]; int[][] g = new int[TEAMS][PLAYERS]; for (i =
   0; i <= TEAMS - 1; i++) {
```

```
      System.out.print("Enter name for team No. " + (i + 1) + ": ");
      t[i] = cin.nextLine();
      for (j = 0; j <= PLAYERS - 1; j++) {
         System.out.print("Enter name of player No. " + (j + 1) + ": ");
         p[i][j] = cin.nextLine();
         System.out.print("Enter goals of player No. " + (j + 1) + ": ");
         g[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
   //Sort array g for (i = 0; i <= TEAMS - 1; i++) {
      for (m = 1; m <= 5; m++) { //Perform 5 passes
         for (n = PLAYERS - 1; n >= m; n--) {
            if (g[i][n] > g[i][n - 1]) {
               temp = g[i][n];
               g[i][n] = g[i][n - 1];
               g[i][n - 1] = temp;
               tempStr = p[i][n];
               p[i][n] = p[i][n - 1];
               p[i][n - 1] = tempStr;
            }
         }
      }
   }
   //Display 5 best scorers of each team for (i = 0; i <= TEAMS - 1; i++) {
      System.out.println("Best scorers of " + t[i]);
      System.out.println("----------------------------------");
      for (j = 0; j <= 4; j++) {
         System.out.println(p[i][j] + " scored " + g[i][j] + " goals");
      }
   }
}
```

Exercise 34.6-7 Counting the Frequency of Vowels

Write a Java program that prompts the user to enter an English sentence and counts the frequency of each vowel in the sentence. Use a hashmap to store the vowels as keys and their frequencies as values.

Solution

In the realm of programming, the manipulation and analysis of textual data play a crucial role. One common task involves counting the frequency of specific elements within a given text, providing insights into its linguistic characteristics. Vowels are fundamental components of the English language, and analyzing their frequency can reveal patterns, aid in language processing, and even assist in certain cryptographic algorithms.

In the solution that follows, the program starts by creating a hashmap named `vowelsFrequency` to store and manage the frequency of each vowel (A, E, I, O,

U), with initial frequencies all set to zero. For each character in the user-provided sentence, the program checks if it is a vowel and, if it is, the corresponding frequency count in the hashmap is updated.

Project_34.6-7

```
public static void main(String[] args) throws Exception {
   int i; String letter;
   //Create a hashmap to store the frequencies of each vowel with initial //frequencies all
   set to zero.
   HashMap<String, Integer> vowelsFrequency = new HashMap<>(
      Map.of("A", 0, "E", 0, "I", 0, "O", 0, "U", 0)
   );
   System.out.print("Enter an English sentence: "); String sentence = cin.nextLine();
   //Iterate through the characters of the user-provided sentence and if it is a vowel,
   //update (increase) the corresponding frequency count in the vowelsFrequency hashmap.
   for (i = 0; i < sentence.length(); i++) {
      letter = ("" + sentence.charAt(i)).toUpperCase();
      if (vowelsFrequency.containsKey(letter)) {
         vowelsFrequency.put(letter, vowelsFrequency.get(letter) + 1);
      }
   }
   //Display the frequencies of each vowel for (String vowel : vowelsFrequency.keySet()) {
      System.out.println(vowel + ": " + vowelsFrequency.get(vowel));
   }
}
```

In order to use the `HashMap` *and* `Map` *classes in your project, you need to import the corresponding libraries using the* `import java.util.HashMap` *and* `import java.util.Map` *statements at the beginning of the project.*

The Java built-in method `struct.containsKey(key_name)` *returns* `true` *when the hashmap* `struct` *contains the specified key* `key_name` *within its keys collection.*

The statement `if (vowelsFrequency.containsKey(letter))` *is equivalent to the statement* `if (vowelsFrequency.containsKey(letter) == true)`**.**

34.7 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) The main idea of the bubble sort algorithm (when sorting an array in ascending order) is to repeatedly move the smallest elements of the array to the lowest index positions.
2) In an array sorted in ascending order, the first element is the greatest of all.
3) When using the bubble sort algorithm, the total number of swaps depends on the given array.

4) The case in which the bubble sort algorithm performs the greatest number of swaps is when you want to sort in descending order an array that is already sorted in ascending order.
5) In the bubble sort algorithm, when the decision control structure tests the Boolean expression `A[n] > A[n − 1]`, it means that the elements of array A are being sorted in descending order.
6) In Java, sorting algorithms compare letters not in the same way that they compare numbers.
7) If you want to sort an array `A` but preserve the one-to-one correspondence with the elements of an array `B`, you must rearrange the elements of array `B` as well.
8) The bubble sort algorithm sometimes performs better than the modified bubble sort algorithm.
9) According to the bubble sort algorithm, in each pass (except the last one) only one element is guaranteed to be placed in proper position.
10) The bubble sort algorithm can be implemented only by using for-loops.
11) The quick sort algorithm cannot be used to sort each column of a two-dimensional array.
12) The insertion sort algorithm can sort in either descending or ascending order.
13) One of the fastest sorting algorithms is the modified bubble sort algorithm.
14) The bubble sort algorithm, for a one-dimensional array of N elements, performs $\frac{N-1}{N}$ compares.
15) The bubble sort algorithm, for a one-dimensional array of N elements, performs $\frac{N(N-1)}{2}$ passes.
16) When using the modified bubble sort algorithm, if a complete pass is performed and no swaps have been done, then the algorithm knows the array is sorted and there is no need for further passes.
17) When using the selection sort algorithm, if you wish to sort an array in descending order, you need to search for maximum values.
18) The selection sort algorithm performs well on computer systems with limited main memory.
19) The selection sort algorithm is suitable for large scale data operations.
20) The selection sort algorithm is a very complex algorithm.

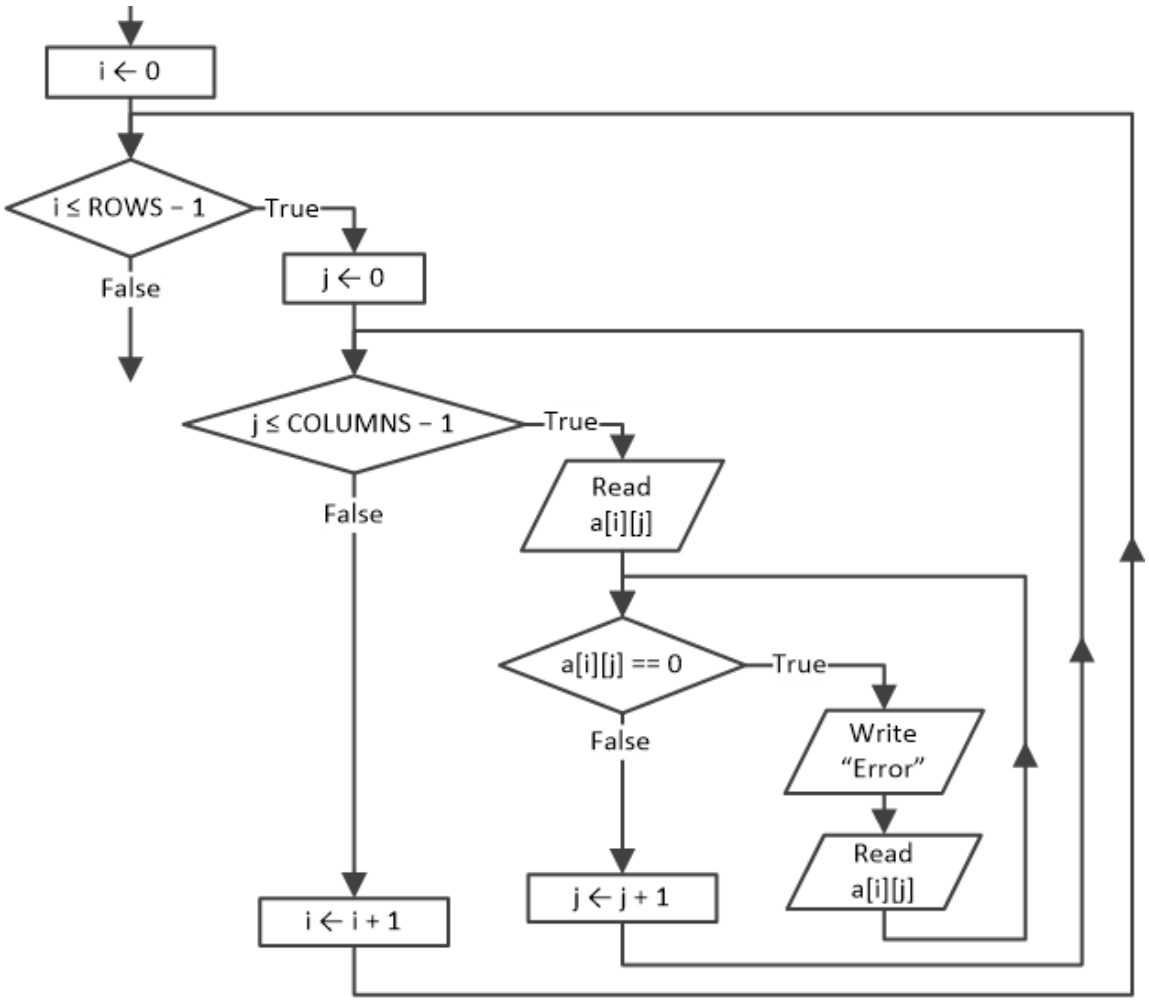
21) The insertion sort algorithm generally performs better than the selection and the bubble sort algorithm.
22) The insertion sort algorithm can sometimes prove even faster than the quicksort algorithm.
23) The quicksort algorithm is considered one of the best and fastest sorting algorithms.
24) A sorted array contains only elements that are different from each other.
25) A search algorithm is an algorithm that searches for an item with specific features within a set of data.
26) The sequential search algorithm can be used only on arrays that contain arithmetic values.
27) One of the most commonly used search algorithms is the quick search algorithm.
28) One search algorithm is called the heap algorithm.
29) A linear (or sequential) search algorithm can work as follows: it can check if the last element of the array is equal to a given value, then it can check the last but one element, and so on, until the beginning of the array or until the given value is found.
30) The linear search algorithm can, in certain situations, find an element faster than the binary search algorithm.
31) The linear search algorithm can be used in large scale data operations.
32) The linear search algorithm cannot be used in sorted arrays.
33) The binary search algorithm can be used in large scale data operations.
34) If an array contains a value multiple times, the binary search algorithm can find only the first in order occurrence of a given value.
35) When using search algorithms, if an array contains unique values and the element that you are looking for is found, there is no need to check any further.
36) The main disadvantage of the binary search algorithm is that data needs to be sorted.
37) The binary search algorithm can be used only in arrays that contain arithmetic values.
38) If the element you are looking for is in the last position of an array, a linear search algorithm that starts searching from the beginning of the array will examine all the elements in the array.
39) The linear search algorithm can be used on two-dimensional arrays.

40) If the element that you are looking for using the binary search algorithm is at the first position of an array with at least three elements, it will be found in just one iteration.

34.8 Review Exercises

Complete the following exercises.

1) Write the Java program that corresponds to the following flowchart fragment.



2) Design a flowchart and write the corresponding Java program that lets the user enter 50 positive numerical values into an array. The algorithm, and consequently the Java program, must then create a new array of 47 elements. In this new array, each position must contain the average value of four elements: the values that exist in the current and the next three positions of the user-provided array.

3) Write a Java program that lets the user enter numerical values into arrays `a`, `b`, and `c`, of 15 elements each. The program must then create a new array

`newArr` of 15 elements. In this new array, each position must contain the lowest value of arrays `a`, `b`, and `c`, for the corresponding position.

Next, design the corresponding flowchart fragment for only that part of your program that creates the array `newArr`.

4) Write a Java program that lets the user enter numerical values into arrays `a`, `b`, and `c`, of 10, 5, and 15 elements respectively. The program must then create a new array `newArr` of 30 elements. In this new array, the first 15 positions must contain the elements of array `c`, the next five positions must contain the elements of array `b`, and the last 10 positions must contains the elements of array `a`.

   Next, design the corresponding flowchart fragment for only that part of your program that creates the array `newArr`.

5) Write a Java program that for two given arrays `a` and `b` of 3 × 4 and 5 × 4 elements respectively it creates a new array `newArr` of 8 × 4 elements. In this new array, the first 3 rows must contain the elements of array `a` and the next 5 rows must contain the elements of array `b`.

6) Write a Java program that lets the user enter numerical values into arrays `a`, `b`, and `c`, of 5 × 10, 5 × 15, and 5 × 20 elements, respectively. The program must then create a new array `newArr` of 5 × 45 elements. In this new array, the first 10 columns must contain the elements of array `a`, the next 15 columns must contain the elements of array `b`, and the last 20 rows must contain the elements of array `c`.

7) Write a Java program that lets the user enter 50 numerical values into an array and then creates two new arrays, `reals` and `integers`. The array `reals` must contain the real values, whereas the array `integers` must contain the integer values. The value 0 (if any) must not be added to any of the final arrays, either `reals` or `integers`.

   Next, design the corresponding flowchart fragment for only that part of your program that creates the arrays `reals` and `integers`.

8) Write a Java program that lets the user enter 50 three-digit integers into an array and then creates a new array containing only the integers in which the first digit is less than the second digit and the second digit is less than the third digit. For example, the values 357, 456, and 159 are such integers.

9) A public opinion polling company asks 200 citizens to each score 10 consumer products. Write a Java program that prompts the user to enter the name of each product and the score each citizen gave (A, B, C, or D). The program must then calculate and display the following: for each product, the name of the product and the number of citizens that gave it an “A”

b) for each citizen, the number of "B" responses they gave c) which product or products are considered the best Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any score with a value other than A, B, C, or D.

10) Write a Java program that prompts the user to enter the names of 20 U.S. cities and the names of 20 Canadian cities and then, for each U.S. city, the distance (in miles) from each Canadian city. Finally, the program must display, for each U.S. city, its closest Canadian city.

11) Design a flowchart and write the corresponding Java program that lets the user enter the names and the heights of 30 mountains, as well as the country in which each one belongs. The algorithm, and consequently the Java program, must then display all available information about the highest and the lowest mountain.

12) Design the flowchart fragment of an algorithm that, for a given array A of N × M elements, finds and displays the maximum value as well as the row and the column in which this value was found.

13) Twenty-six teams participate in a football tournament. Each team plays 15 games, one game each week. Write a Java program that lets the user enter the name of each team and the letter "W" for win, "L" for loss, and "T" for tie (draw) for each game. If a win receives 3 points and a tie 1 point, the Java program must find and display the name of the team that wins the championship based on which team obtained the greatest sum of points. Assume that no two teams have an equal sum of points.

14) On Earth, a free-falling object has an acceleration of 9.81 m/s^2 downward. This value is denoted by g. A student wants to calculate that value using an experiment. She allows 10 different objects to fall downward from a known height, and measures the time they need to reach the floor. However, since her chronometer is not so accurate, she does this 20 times for each object. She needs a Java program that allows her to enter the heights (from which objects are left to fall), as well as the measured times that they take to reach the floor. The program must then
► calculate g and store all calculated values in a 10 × 20 array.
► find and display the minimum and the maximum calculated values of g for each object.

- find and display the overall minimum and maximum calculated values of *g* of all objects.

The required formula is $S = u_o + \frac{1}{2}at^2$

where

- ***S*** is the distance that the free-falling objects traveled, in meters (m) ***u_o*** is the initial velocity (speed) of the free-falling objects in meters per second (m/sec). However, since the free-falling objects start from rest, the value of u_0 must be zero.
- ***t*** is the time that it took the free-falling object to reach the floor, in seconds (sec) ***g*** is the acceleration, in meters per second^2 (m/sec^2) 15) Ten measuring stations, one in each city, record the daily CO_2 levels for a period of a year. Write a Java program that lets the user enter the name of each city and the CO_2 levels recorded at the same hour each day. The Java program then displays the name of the city that has the clearest atmosphere (on average).

16) Design the flowchart fragment of an algorithm that, for a given array A of N × M elements, finds and displays the minimum and the maximum values of each row.

17) Write a Java program that lets the user enter values into a 20 × 30 array and then finds and displays the minimum and the maximum values of each column.

18) Twenty teams participate in a football tournament, and each team plays 10 games, one game each week. Write a Java program that prompts the user to enter the name of each team and the letter "W" for win, "L" for loss, and "T" for tie (draw) for each game. If a win receives 3 points and a tie 1 point, the Java program must find and display the names of the teams that win the gold, the silver, and the bronze medals based on which team obtained the greatest sum of points. Use the modified bubble sort algorithm. Assume that no two teams have an equal sum of points.

    Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any letter other than W, L, or T.

    Hint: Instead of performing 19 passes (there are 20 teams), given that only the three best teams must be found, the modified bubble

sort algorithm can perform just 3 passes.

19) Write a Java program that prompts the user to enter the names and the heights of 50 people. The program must then display this information, sorted by height, in descending order. In cases where two or more people share the same height, their names must be displayed in alphabetical order. To achieve this, use the bubble sort algorithm, adapted accordingly.

20) Write a Java program that prompts the user to enter the first names, last names and father's names of 50 people. The program must then display this information, sorted by last name. In cases where two or more people share the same last name, their first names must be displayed in alphabetical order. Additionally, if two or more people share the same first name, their father's names must be displayed in alphabetical order. To achieve this, use the bubble sort algorithm, adapted accordingly.

21) In a song contest there are 10 judges, each of whom scores 12 artists for their performance. However, according to the rules of this contest, the total score is calculated after excluding the highest and lowest scores. Write a Java program that prompts the user to enter the names of the artists and the score they get from each judge. The program must then display

a) for each artist, their name and total score, after excluding the maximum and the minimum scores. Assume that each artist's highest and lowest scores are unique, meaning they won't have multiple scores with the same value.

b) the final classification, starting with the artist that has the greatest score. However, if two or more artists have the same score, their names must be displayed in alphabetical order. Use the bubble sort algorithm, adapted accordingly.

22) Design the flowchart fragment of an algorithm that, for a given array A of 20 × 8 elements, sorts each row in descending order using the bubble sort algorithm. Assume that the array contains numerical values.

23) Design the flowchart fragment of an algorithm that, for a given array A of 5 × 10 elements, sorts each column in ascending order using the bubble sort algorithm. Assume that the array contains numerical values.

24) Design the flowchart fragment of an algorithm that, for a given array A of 20 × 8 elements, sorts each row in descending order

using the insertion sort algorithm. Assume that the array contains numerical values.

25) Design the flowchart fragment of an algorithm that, for a given array `A` of 5 × 10 elements, sorts each column in ascending order using the selection sort algorithm. Assume that the array contains numerical values.

26) In a Sudoku contest, 10 participants compete to solve eight different Sudoku puzzles as quickly as possible. Write a Java program that lets the user enter the name of each contestant and their time (in hours, minutes and seconds) to complete each puzzle. The program must then display for each contestant, their name along with their three best times. Assume that the times of each contestant are different.

    a) the names of the three contestants who receive the gold, the silver, and the bronze medals based on the contestant with the lowest average time. Assume that no two contestants have the same average time.

    Use the selection sort algorithm when necessary.

    Hint: Given that only the three best contestants must be found, the selection sort algorithm could sort only the first three elements.

27) Five measuring stations, one in each area of a large city, record the daily carbon dioxide (CO_2) levels on an hourly basis. Write a Java program that lets the user enter the name of each area and the CO_2 levels recorded every hour (00:00 to 23:00) for a period of two days. The Java program then must calculate and display for each

    a) area, its name and its average CO_2 level
    b) for each hour, the average CO_2 level of the city
    c) the hour in which the city atmosphere was most polluted (on average)
    d) the hour and the area in which the highest level of CO_2 was recorded
    e) the three areas with the dirtiest atmosphere (on average), using the insertion sort algorithm

28) Design the flowchart fragment of the linear search algorithm that performs a search on array `a` of N elements to find the value `needle` and displays the position index(es) at which `needle` is found. If `needle` is not found, the message “Not found” must be displayed. Assume that the array contains numerical values.

29) Design the flowchart fragment of the binary search algorithm that performs a search on array `a` of N elements to find the value `needle` and displays the position at which `needle` is found. If

`needle` is not found, the message “Not found” must be displayed. Assume that the array contains numerical values.

30) Twelve teams participate in a football tournament, and each team plays 20 games, one game each week. Write a Java program that prompts the user to enter the name of each team and the letter “W” for win, “L” for loss, or “T” for tie (draw) for each game. Then the program must prompt the user for a letter (W, L, or T) and display, for each team, the week number(s) in which the team won, lost, or tied respectively. For example, if the user enters “L”, the Java program must search and display, for each team, the week numbers (e.g., Week 3, Week 14, and so on) in which the team lost the game.

31) Ten teams participate in a football tournament, and each team plays 16 games, one game each week. Write a Java program that prompts the user to enter the name of each team, the number of goals the team scored, and the number of goals the team let in for each match. A win receives 3 points and a tie receives 1 point. The Java program must then prompt the user for a team name and finally calculate and display the total number of points for this team. If the user-provided team name is not found, the message “This team does not exist” must be displayed.

    Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any negative number of goals.

    Assume that no two teams share the same name.

32) In a high school, there are two classes, with 20 and 25 students respectively. Write a Java program that prompts the user to enter the names of the students in two separate arrays. The program then displays the names of each class independently in ascending order. Afterwards, the program prompts the user to enter a name and it searches for that user-provided name in both arrays. If the student's name is found, the program must display the message “Student found in Class No N”, where N can be either 1 or 2; otherwise the message “Student not found in either class” must be displayed. Assume that both arrays contain unique names.

    Hint: Since the arrays are sorted and the names are unique, use the binary search algorithm.

33) Suppose there are two arrays, `usernames` and `passwords`, that contain the login information of 100 employees of a company.

Write a code fragment that prompts the user to enter a username and a password and then displays the message “Login OK!” when the combination of username and password is valid; the message “Login Failed!” must be displayed otherwise. Both usernames and passwords are case-insensitive. Assume that usernames are unique but passwords are not.

34) Suppose there are two arrays, `names` and `SSNs`, that contain the names and the SSNs (Social Security Numbers) of 1,000 U.S. citizens. Write a code fragment that prompts the user to enter a value (it can be either a name or an SSN) and then searches for and displays the names of all the people that have this name or this SSN. If the user-provided value is not found, the message “This value does not exist” must be displayed.

35) There are 12 students and each one of them has received their grades for six lessons. Write a Java program that lets the user enter the grades for all lessons and then displays a message indicating whether or not there is at least one student that has an average value below 70. Moreover, using a loop control structure, the program must validate data input and display a different error message for each type of input error when the user enters any negative value, or a value greater than 100.

36) Write a Java program that prompts the user to enter an English message, and then, using the table that follows, displays the corresponding Morse code using dots and dashes. Please note that space characters must be displayed as slash characters (/) in the translated message.

Morse Code			
A	.-	N	-.
B	-...	O	---
C	-.-.	P	.--.
D	-..	Q	--.-
E	.	R	.-.
F	..-.	S	...
G	--.	T	-
H	....	U	..-

I	..	V	...-
J	.---	W	.--
K	-.-	X	-..-
L	.-..	Y	-.--
M	--	Z	--..

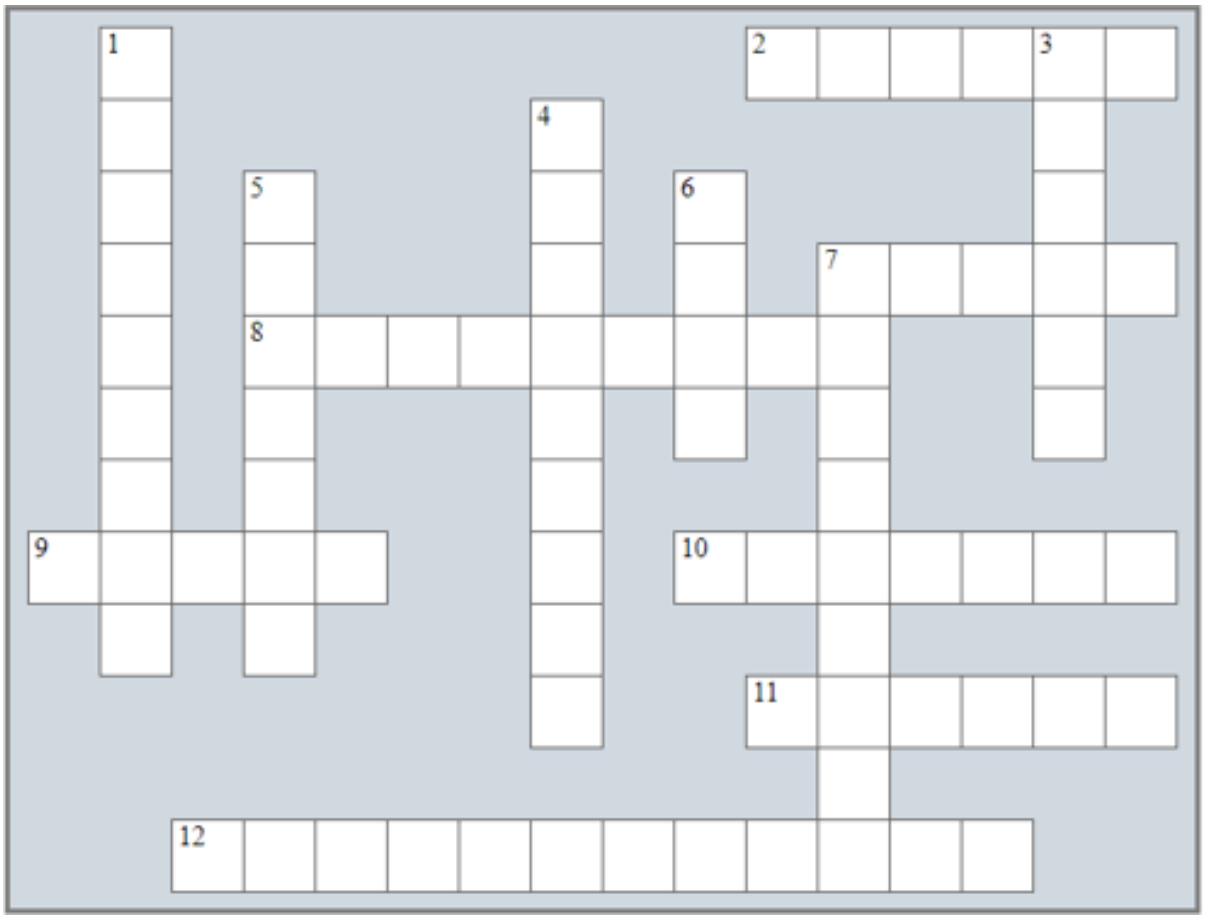
Hint: Use a hashmap to hold the Morse code.

37) Write a Java program that prompts the user to enter an English sentence. The program must then display: a) all letters that exist in the user-provided sentence along with their frequency count b) all letters that do not exist in the user-provided sentence c) the percentage of letters that do not exist in relation to the letters of the English alphabet d) the percentage of non-alphabetic characters in relation to the characters of the user-provided sentence (excluding space characters) Hint: Use a hashmap to store all 26 English letters as keys and their frequencies as values, but since Java's `map.of()` method supports only up to 10 key-value pairs, you need to find another way to create it.

Review in “Data Structures in Java”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

2) A sorting algorithm.

7) Each array element is assigned a unique number known as an _____.

8) This sorting algorithm performs well on computer systems in which limited main memory (RAM) comes into play.

9) A mutable data structure in Java.

10) The process of putting the elements of an array in a certain order.

11) A search algorithm.

12) In a square matrix, the collection of those elements that runs from the top right corner to the bottom left corner.

Down

1) A data _____ is a collection of data organized so that you can perform operations on it in the most effective way.

3) Another name for the sequential search algorithm.

4) It is considered one of the best and fastest sorting algorithms.

5) Its elements can be uniquely identified using a key and not necessarily an integer value.

6) In this diagonal, the elements have their row index equal to their column index.

7) This sorting algorithm can prove very fast when sorting very small arrays— sometimes even faster than the quicksort algorithm.

Review Questions

Answer the following questions.

1) What limitation do variables have that arrays don't?
2) What is a data structure?
3) What is each item of a data structure called?
4) Name six known data structures that Java supports.
5) What is an array in Java?
6) What is a hashmap in Java?
7) What does it mean when we say that an array is “mutable”?
8) What happens when a statement tries to display the value of a non-existing array element?
9) What happens when a statement tries to assign a value to a non-existing hashmap element?
10) In an array of 100 elements, what is the index of the last element?
11) What does “iterating through rows” mean?
12) What does “iterating through columns” mean?
13) What is a square matrix?

14) What is the main diagonal of a square matrix?
15) What is the antidiagonal of a square matrix?
16) Write the code fragment in general form that validates data input to an array without displaying any error messages.
17) Write the code fragment in general form that validates data input to an array and displays a generic error message (that is, the same error message for any type of input error).
18) Write the code fragment in general form that validates data input to an array and displays a different error message for each type of input error.
19) What is a sorting algorithm? Name five sorting algorithms.
20) Which sorting algorithm is considered the most inefficient?
21) Can a sorting algorithm be used to find the minimum or the maximum value of an array?
22) Why is a sorting algorithm not the best option to find the minimum or the maximum value of an array?
23) Write the code fragment that sorts array `a` of N elements in ascending order, using the bubble sort algorithm. Assume that the array contains numerical values.
24) For a given array of N elements, how many compares does the bubble sort algorithm perform?
25) When does the bubble sort algorithm perform the maximum number of swaps?
26) Using the bubble sort algorithm, write the code fragment that sorts array `a` but preserves the one-to-one correspondence with the elements of array `b` of N elements in ascending order. Assume that the array contains numerical values.
27) Using the modified bubble sort algorithm, write the code fragment that sorts array `a` of N elements in ascending order. Assume that the array contains numerical values.
28) Using the selection sort algorithm, write the code fragment that sorts array `a` of N elements in ascending order. Assume that the array contains numerical values.

29) Using the insertion sort algorithm, write the code fragment that sorts array `a` of N elements in ascending order. Assume that the array contains numerical values.

30) What is a search algorithm? Name the two most commonly used search algorithms.

31) What are the advantages and disadvantages of the linear search algorithm?

32) Using the linear search algorithm, write the code fragment that performs a search on array `a` to find value `needle`. Assume that the array contains numerical values.

33) What are the advantages and disadvantages of the binary search algorithm?

34) Using the binary search algorithm, write the code fragment that performs a search on array `a` to find value `needle`. Assume that the array contains numerical values and is sorted in ascending order.

Part VII
Subprograms

Chapter 35
Introduction to Subprograms

35.1 What Exactly is a Subprogram?

In computer science, a *subprogram* is a block of statements packaged as a unit that performs a specific task. A subprogram can be called several times within a program, whenever that specific task needs to be performed.

In Java, a *built-in method* is an example of such a subprogram. Take the well-known `Math.abs()` method, for example. It consists of a block of statements packaged as a unit under the name “abs”, and it performs a specific task—it returns the absolute value of a number.

✎ *If you are wondering what kind of statements might exist inside the method* `Math.abs()`*, here is a possible block of statements.*

```
if (number < 0)
   return number * (-1);
else
   return number;
```

Generally speaking, there are two kinds of subprograms: *functions* and *procedures*. The difference between a function and a procedure is that a function returns a result, whereas a procedure doesn't. However, in some computer languages, this distinction may not quite be apparent. There are languages in which a function can also behave as a procedure and return no result, and there are languages in which a procedure can return one or even more than one result.

✎ *Depending on the computer language being used, the terms “function” and “procedure” may be different. For example, in Visual Basic you can find them as “functions” and “subprocedures”, in FORTRAN as “functions” and “subroutines”, whereas in Java, the preferred terms are usually “methods” and “void methods”.*

35.2 What is Procedural Programming?

Suppose you were assigned a project to solve the drug abuse problem in your area. One possible approach (which could prove very difficult or even impossible) would be to try to solve this problem by yourself!

A better approach, however, would be to subdivide the large problem into smaller subproblems such as prevention, treatment, and rehabilitation, each of which could be further subdivided into even smaller subproblems, as shown in **Figure 35–1**.

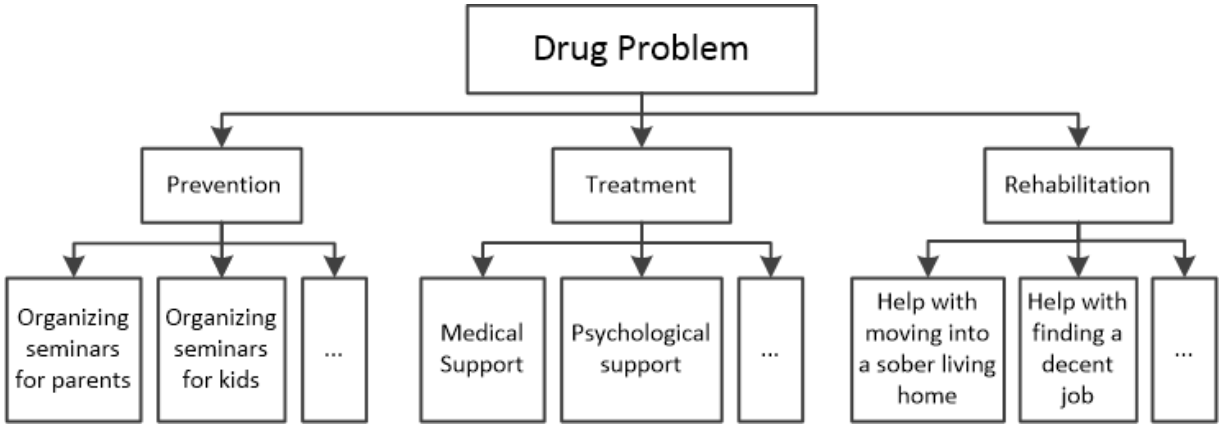


Figure 35–1 A problem can be subdivided into smaller problems

As the supervisor of this project, you could rent a building and establish within it three departments: the prevention department, with all of its subdepartments; the treatment department, with all of its subdepartments; and the rehabilitation department with all of its subdepartments. Finally, you would hire staff (specialists from a variety of fields), you would build teams and employ them to do the job for you!

Procedural programming does exactly the same thing. It subdivides an initial problem into smaller subproblems, and each subproblem is further subdivided into smaller subproblems. Finally, for each subproblem a small subprogram is written, and the main program (as does the supervisor), calls (employs) each of them to do a different part of the job.

Procedural programming offers several advantages:

- It enables programmers to reuse the same code whenever necessary, without the need for rewriting or copying it.
- It is relatively easy to implement.
- It helps programmers follow the flow of execution more easily, simplifying the debugging process.

✎ *A very large program can prove very difficult to debug and maintain when it is all in one piece. For this reason, it is often easier to subdivide it into smaller subprograms, each of which performs a clearly defined process.*

35.3 What is Modular Programming?

In *modular programming*, subprograms of common functionality can be grouped together into separate modules, and each module can have its own set of data. Therefore, a program can consist of more than one part, and each of those parts (modules) can contain one or more smaller parts (subprograms).

✎ *The* `Math` *module is such an example. It contains subprograms of common functionality* (*related to Math*), *such as* `abs()`, `sqrt()`, `sin()`, `cos()`, `tan()`, *and many more.*

If you were to use modular programming in the previous drug problem example, then you could have three separate buildings—one to host the prevention department and all of its subdepartments, a second one to host the treatment department and all of its subdepartments, and a third one to host the rehabilitation department and all of its subdepartments (as shown in **Figure 35–2**). These three buildings could be thought of as three different modules in modular programming, each of which would contain subprograms of common functionality.

Figure 35–2 Subprograms of common functionality can be grouped together into separate modules.

35.4 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) A subprogram is a block of statements packaged as a unit that performs a specific task.
2) In general, there are two kinds of subprograms: functions and procedures.
3) In general, the difference between a function and a procedure is that a procedure returns a result, whereas a function does not.

4) Java supports only procedures.
5) Procedural programming subdivides the initial problem into smaller subproblems.
6) An advantage of procedural programming is the ability to reuse the same code, without the need for rewriting or copying it.
7) Procedural programming helps programmers follow the flow of execution more easily.
8) Modular programming increases program development speed.
9) In modular programming, subprograms of common functionality are grouped together into separate modules.
10) In modular programming, each module can have its own set of data.
11) Modular programming uses different structures than structured programming does.
12) A program can consist of more than one module.

Chapter 36
User-Defined Subprograms

36.1 Subprograms that Return a Value

In many computer languages, a subprogram that returns a value is called a *function.* Java calls them *methods* and there are two categories of methods. There are the *built-in methods*, such as `Math.abs()`, `Math.sqrt()`, and there are the *user-defined methods*, those that you can personally write and use in your own programs.

The general form of a Java method that returns a value is shown here.

```
static return_type name([type1 arg1, type2 arg2, type3 arg3, …]) {
    Local variables declaration section

    A statement or block of statements

    return value; }
```

where

- *return_type* is the data type of the value that the method returns.
- *name* is the name of the method.
- *arg1*, *arg2*, *arg3*, … is a list of arguments (variables, arrays etc.) used to pass values from the caller to the method. There can be as many arguments as you need.
- *type1*, *type2*, *type3*, … is the data type of each argument. Each argument must have a data type.
- *value* is the value returned to the caller. It can be a constant value, a variable, an expression, or even a data structure. Its data type must match the *return_type* of the method.

✎ *Note that arguments are optional; that is, a method may contain no arguments.*

✎ *In Section 5.4 you learned about the rules that must be followed when assigning names to variables. Assigning names to subprograms follows exactly the same rules!*

The method *name* can be likened to a box (see **Figure 36–1**) which contains a statement or block of statements. It accepts the arguments *arg1*, *arg2*, *arg3*, … as input values and returns *value* as output value.

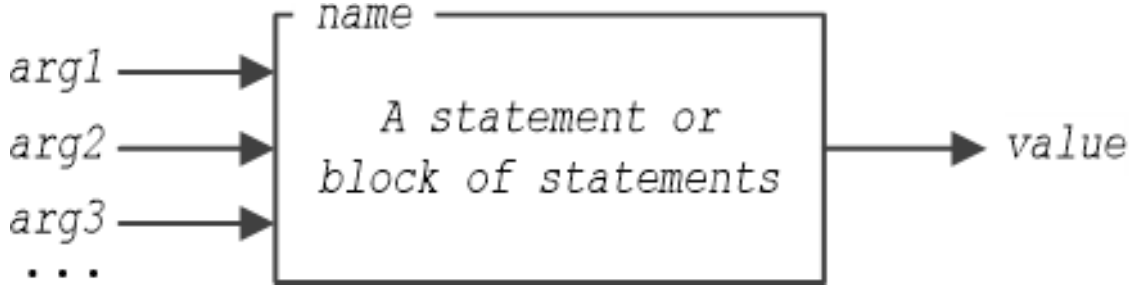


Figure 36–1 A method can be likened to a box For example, the next method accepts two numbers through the arguments num1 and num2, then calculates their sum and returns the result.

```
static double getSum(double num1, double num2) {
   double result;
   result = num1 + num2; return result; }
```

Of course, this can also be written as

```
static double getSum(double num1, double num2) {
    return num1 + num2; }
```

36.2 How to Make a Call to a Method

Every call to a method is as follows: you write the name of the method followed by a list of arguments (if required), either within a statement that assigns the method's returned value to a variable or directly within an expression.

Let's see some examples. The following method accepts a numeric value through the argument num and returns the result of that value raised to the power of three.

```
static double cube(double num) {
   double result;
   result = num * num * num; return result; }
```

Now, suppose that you want to calculate a result of the following expression $y = x^3 + \frac{1}{x}$

You can either assign the returned value from the method cube() to a variable, as shown here

```
x = Double.parseDouble(cin.nextLine());
```

cb = **cube**(x); //*Assign the returned value to a variable* y = cb + 1 / x; //*and use that variable*
System.out.**println**(y);

or you can call the method directly in an expression,

x = Double.**parseDouble**(cin.**nextLine**());
y = **cube**(x) + 1 / x; //*Call the method directly in an expression*
System.out.**println**(y);

or you can even call the method directly in a `System.out.println()` statement.

```
x = Double.parseDouble(cin.nextLine()); System.out.println(cube(x) + 1 / x); //Call
the method directly
//in a System.out.println() statement
```

> ✎ *User-defined methods can be called just like the built-in methods of Java.*

Now let's see another example. The next Java program defines the method `getMessage()` and then the main code calls it. The returned value is assigned to variable a.

```
🗀 Project_36.2a
//Define the method static String getMessage() {
                    String msg;
          msg = "Hello Zeus"; return msg; }
   //Main code starts here public static void
     main(String[] args) throws Exception {
                    String a;
 System.out.println("Hi there!"); a = getMessage();
              System.out.println(a); }
```

If you run this program, the following messages are displayed.

✎ *Note that subprograms must be written outside of the main method.*

✎ *A method does not execute immediately when a program starts running. The first statement that actually executes in the last example is the statement* `System.out.println("Hi there!")`.

You can pass (send) values to a method, as long as at least one argument exists within the method's parentheses. In the next example, the method `display()` is called three times but each time a different value is passed through the argument `color`.

```
                    Project_36.2b
//Define the method static String display(String color)
                              {
                          String msg;
      msg = "There is " + color + " in the rainbow";
                        return msg; }
       //Main code starts here public static void
         main(String[] args) throws Exception {
            System.out.println(display("red"));
          System.out.println(display("yellow"));
          System.out.println(display("blue")); }
```

If you run this program, the following messages are displayed.

In the next example, two values must be passed to method `display()`.

```
Project_36.2c
static String display(String color, boolean exists) {
                        String neg;
                         neg = "";
                       if (!exists) {
                        neg = "n't any";
                             }
    return "There is" + neg + " " + color + " in the
                        rainbow"; }
public static void main(String[] args) throws Exception
                             {
        System.out.println(display("red", true));
       System.out.println(display("yellow", true));
      System.out.println(display("black", false)); }
```

If you run this program the following messages are displayed.

✎ *In Java, you can place your methods either above or below your main code. Most programmers, though, prefer to have them all on the top for better observation.*

36.3 Subprograms that Return no Values

In computer science, a subprogram that returns no values can be known as a procedure, subprocedure, subroutine, void function, and more. In Java, the preferred term is usually *void method*.

The general form of a Java void method is

static void *name*(*[**type1** arg1, **type2** arg2, **type3** arg3, …]*) {

```
Local variables declaration section
```

```
A statement or block of statements
```

}

where

- `name` is the name of the void method.
- `arg1`, `arg2`, `arg3`, … is a list of arguments (variables, arrays etc.) used to pass values from the caller to the void method. There can be as many arguments as you want.
- `type1`, `type2`, `type3`, … is the data type of each argument. Each argument must have a data type.

✎ *Note that arguments are optional; that is, a void method may contain no arguments.*

For example, the next void method accepts two numbers through the arguments `num1` and `num2`, then calculates their sum and finally displays the result.

```
static void displaySum(double num1, double num2) {
   double result;
   result = num1 + num2; System.out.println(result); }
```

36.4 How to Make a Call to a void Method

You can make a call to a void method by just writing its name. The next example defines the void method `displayLine()` and the main code calls the void method whenever it needs to display a horizontal line.

🗀 **Project_36.4a**

```
//Define the void method static void displayLine() {
    System.out.println("----------------------"); }
public static void main(String[] args) throws Exception
                                {
   System.out.println("Hello there!"); displayLine();
  System.out.println("How do you do?"); displayLine();
      System.out.println("What is your name?");
                   displayLine(); }
```

You can pass (send) values to a void method, as long as at least one argument exists within void method's parentheses. In the next example, the void method `displayLine()` is called three times but each time a different value is passed through the variable `length`, resulting in three printed lines of different length.

🗀 **Project_36.4b**

```
static void displayLine(int length) {
                    int i;
          for (i = 1; i <= length; i++) {
                System.out.print("-");
                    }
              System.out.println(); }
```

```
public static void main(String[] args) throws Exception
                                {
   System.out.println("Hello there!"); displayLine(12);
          System.out.println("How do you do?");
     displayLine(14); System.out.println("What is your
                 name?"); displayLine(18); }
```

Since the void method `displayLine()` *returns no value, the following line of code is* ***wrong.*** *You* ***cannot*** *assign the void method to a variable because there isn't any returned value!*

```
y = display_line(12);
```

Also, you ***cannot*** *call it within a statement. The following line of code is also* ***wrong****.*

```
System.out.println("Hello there!\n" + display_line(12));
```

36.5 Formal and Actual Arguments

Each method (or void method) contains an argument list called a *formal argument list*. As already stated, arguments in this list are optional; the formal argument list may contain no arguments, one argument, or more than one argument.

When a subprogram (method, or void method) is called, an argument list may be passed to the subprogram. This list is called an *actual argument list*.

In the next example, the formal arguments (variables) `n1` and `n2` constitute the formal argument list whereas the formal arguments (variables) `x` and `y`, as well as formal arguments (expressions) `x + y` and `y / 2`, constitute the actual argument lists.

Project_36.5

```
//Define the method multiply().
//The two arguments n1 and n2 are called formal arguments.
static double multiply( double n1, double n2 ) { [More…]
   double result;
   result = n1 * n2; return result; }
//Main code starts here public static void main(String[] args) throws Exception {
   double x, y, w;
   x = Double.parseDouble(cin.nextLine()); y = Double.parseDouble(cin.nextLine());
```

```
    //Call the method multiply().
    //The two arguments x and y are called actual arguments.
    w = multiply( x, y ); [More…]
    System.out.println(w);
    //Call the method multiply().
    //The two arguments x + y and y / 2 are called actual arguments.
    System.out.println(multiply( x + 2, y / 2 )); [More…]
}
```

✎ *Note that there is a one-to-one correspondence between the formal and the actual arguments. In the first call, the value of the actual argument* `x` *is passed to the formal argument* `n1`*, and the value of actual argument* `y` *is passed to the formal argument* `n2`*. In the second call, the value of the actual argument (the result of the expression)* `x + 2` *is passed to the formal argument* `n1`*, and the value of the actual argument (the result of the expression)* `y / 2` *is passed to the formal argument* `n2`*.*

36.6 How Does a Method Execute?

When the main code calls a method the following steps are performed:

- The execution of the statements of the main code is interrupted.
- The values of the variables or the result of the expressions that exist in the actual argument list are passed (assigned) to the corresponding arguments (variables) in the formal argument list, and the flow of execution goes to where the method is written.
- The statements of the method are executed.
- When the flow of execution reaches a `return` statement, a value is returned from the method to the main code and the flow of execution continues from where it was before calling the method.

In the next Java program, the method `maximum()` accepts two arguments (numeric values) and returns the greater of the two values.

Project_36.6

```
static double maximum(double val1, double val2) {
    double m;
    m = val1;
    if (val2 > m) {
        m = val2;
```

```
   }
   return m; }
//Main code starts here public static void main(String[] args) throws Exception {
   double a, b, maxim;
   a = Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine());
   maxim = maximum(a, b); System.out.println(maxim); }
```

When the Java program starts running, the first statement executed is the statement `a = Double .parseDouble(cin.nextLine())` (this is considered the first statement of the program).

Below is a trace table that shows the exact flow of execution, how the values of the variables `a` and `b` are passed from the main code to the method, and how the method returns its result. Suppose the user enters the values 3 and 8.

Step	Statements of the Main Code	a	b	maxim
1	`a = Double.parseD…`	**3.0**	?	?
2	`b = Double.parseD…`	3.0	**8.0**	?
3	`maxim = maximum(a, b)`			

When the call to the method `maximum()` is made, the execution of the statements of the main code is interrupted, the values of the variables `a` and `b` are passed (assigned, if you prefer) to the corresponding formal arguments (variables) `val1` and `val2`, and the flow of execution goes to where the method is written. Then the statements of the method are executed.

Step	Statements of Method `maximum()`	val1	val2	m
4	`m = val1`	3.0	8.0	**3.0**
5	`if (val2 > m)`	This evaluates to `true`		
6	`m = val2`	3.0	8.0	**8.0**
7	`return m`			

When the flow of execution reaches the `return` statement, the value 8 is returned from the method to the main code (and assigned to the variable `maxim`) and the flow of execution continues from where it was before

calling the method. The main code displays the value of 8 on the user's screen.

Step	Statements of the Main Code	a	b	maxim
8	`.println(maxim)`	3.0	8.0	**8.0**

Exercise 36.6-1 Back to Basics – Calculating the Sum of Two Numbers

Do the following: i) Write a subprogram named `total` *that accepts two numeric values through its formal argument list and then calculates and returns their sum.*

ii) Using the subprogram cited above, write a Java program that lets the user enter two numbers and then displays their sum. Next, create a trace table to determine the values of the variables in each step of the Java program for two different executions.

The input values for the two executions are: (i) 2, 4; and (ii) 10, 20.

Solution In this exercise you need to write a method that accepts two values from the caller (this is the main code) and then calculates and returns their sum. The solution is shown here.

Project_36.6-1

```
static double total(double a, double b) {
   double s;
   s = a + b;
   return s; }
public static void main(String[] args) throws Exception {
   double num1, num2, result;
   num1 = Double.parseDouble(cin.nextLine()); num2 =
   Double.parseDouble(cin.nextLine());
   result = total(num1, num2); System.out.println("The sum of " + num1 + " + " + num2
   + " is " + result); }
```

Now, let's create the corresponding trace tables. Since you have become more experienced with them, the column “Notes” has been removed.

i) For the input values of 2, 4, the trace table looks like this.

Step	Statement	Main Code			Method `total()`		
		num1	num2	result	a	b	s

1	`num1 = Double.parseD…`	**2.0**	?	?			
2	`num2 = Double.parseD…`	2.0	**4.0**	?			
3	`result = total(num1, num2)`				**2.0**	**4.0**	?
4	`s = a + b`				2.0	4.0	**6.0**
5	`return s`	2.0	4.0	**6.0**			
6	`.println("The sum of "+ …`	It displays: The sum of 2.0 + 4.0 is 6.0					

ii) For the input values of 10, 20, the trace table looks like this.

Step	Statement	Main Code			Method `total()`		
		num1	num2	result	a	b	s
1	`num1 = Double.parseD…`	**10.0**	?	?			
2	`num2 = Double.parseD…`	10.0	**20.0**	?			
3	`result = total(num1, num2)`				**10.0**	**20.0**	?
4	`s = a + b`				10.0	20.0	**30.0**
5	`return s`	10.0	20.0	**30.0**			
6	`.println("The sum of "+ …`	It displays: The sum of 10.0 + 20.0 is 30.0					

Exercise 36.6-2 Calculating the Sum of Two Numbers Using Fewer Lines of Code!

Rewrite the Java program of the previous exercise using fewer lines of code.

Solution The solution is shown here.

Project_36.6-2

```
static double total(double a, double b) {
   return a + b; }
```

```
public static void main(String[] args) throws Exception {
   double num1 = Double.parseDouble(cin.nextLine()); double num2 =
   Double.parseDouble(cin.nextLine());
   System.out.println("The sum of " + num1 + " + " + num2 + " is " + total(num1,
   num2)); }
```

Contrary to the solution of the previous exercise, in this method `total()`, the sum is not assigned to variable `s` but is directly calculated and returned. Furthermore, in this main code, variables `num1` and `num2` are directly declared when first used, whereas the returned value in not assigned to a variable but is directly displayed.

User-defined methods can be called just like the built-in methods of Java.

36.7 How Does a void Method Execute?

When the main code calls a void method, the following steps are performed: ► The execution of the statements of the main code is interrupted.

- The values of the variables or the result of the expressions that exist in the actual argument list are passed (assigned) to the corresponding arguments (variables) in the formal argument list and the flow of execution goes to where the void method is written.
- The statements of the void method are executed.
- When the flow of execution reaches the end of the void method, the flow of execution continues from where it was before calling the void method.

In the next Java program, the void method `minimum()` accepts three arguments (numeric values) through its formal argument list and displays the lowest value.

Project_36.7

```
static void minimum(double val1, double val2, double val3) {
   double minim;
   minim = val1; if (val2 < minim) {
      minim = val2;
   }
   if (val3 < minim) {
      minim = val3;
```

```
    }
    System.out.println(minim); }
public static void main(String[] args) throws Exception {
    double a, b, c;
    a = Double.parseDouble(cin.nextLine()); b = Double.parseDouble(cin.nextLine()); c
    = Double.parseDouble(cin.nextLine());
    minimum(a, b, c);
    System.out.println("The end"); }
```

When the Java program starts running, the first statement executed is the statement `a = Double .parseDouble(cin.nextLine())` (this is considered the first statement of the program). Suppose the user enters the values 9, 6, and 8.

Step	Statements of the Main Code	a	b	c
1	`a = Double.parseD…`	**9.0**	?	?
2	`b = Double.parseD…`	9.0	**6.0**	?
3	`c = Double.parseD…`	9.0	6.0	**8.0**
4	`minimum(a, b, c)`			

When a call to the void method `minimum()` is made, the execution of the statements of the main code is interrupted, the values of the variables `a`, `b`, and `c` are passed to the corresponding formal arguments `val1`, `val2`, and `val3`, and the statements of the void method are executed.

Step	Statements of void Method `minimum()`	val1	val2	val3	minim
5	`minim = val1`	9.0	6.0	8.0	**9.0**
6	`if (val2 < minim)`	This evaluates to `true`			
7	`minim = val2`	9.0	6.0	8.0	**6.0**
8	`if (val3 < minim)`	This evaluates to `false`			
9	`.println(minim)`	It displays: 6.0			

When the flow of execution reaches the end of the void method the flow of execution simply continues from where it was before calling the void method.

Step	Statements of the Main Code	a	b	c

10	`.println("The end")`	It displays: The end

✎ *Note that between step 9 and step 10, no values are returned from the void method to the main code.*

Exercise 36.7-1 Back to Basics – Displaying the Absolute Value of a Number

Do the following: i) Write a subprogram named `displayAbs` *that accepts a numeric value through its formal argument list and then displays its absolute value. Do not use the built-in* `Math.abs()` *method of Java.*

ii) Using the subprogram cited above, write a Java program that lets the user enter a number and then displays its absolute value followed by the user-provided value. Next, create a trace table to determine the values of the variables in each step of the Java program for two different executions.

The input values for the two executions are: (i) 5, and (ii) −5.

Solution In this exercise you need to write a void method that accepts a value from the caller (this is the main code) and then calculates and displays its absolute value. The solution is shown here.

Project_36.7-1

```
static void displayAbs(double n) {
   if (n < 0) {
      n = (-1) * n;
   }
   System.out.println(n); }
public static void main(String[] args) throws Exception {
   double a;
   a = Double.parseDouble(cin.nextLine()); displayAbs(a); //This displays the
   absolute value of the user-provided number.
   System.out.println(a); //This displays the user-provided number.
}
```

Now, let's create the corresponding trace tables.

i) For the input value of 5, the trace table looks like this.

Step	Statement	Main Code	void Method `displayAbs()`

		a	n
1	`a = Double.parseD…`	**5.0**	
2	`displayAbs(a)`		**5.0**
3	`if (n < 0)`	This evaluates to `false`	
4	`.println(n)`	It displays: 5.0	
5	`.println(a)`	It displays: 5.0	

ii) For the input value of −5, the trace table looks like this.

Step	Statement	Main Code	void Method `displayAbs()`
		a	n
1	`a = Double.parseD…`	**−5.0**	
2	`displayAbs(a)`		**−5.0**
3	`if (n < 0)`	This evaluates to `true`	
4	`n = (-1) * n`		**5.0**
5	`.println(n)`	It displays: 5.0	
6	`.println(a)`	It displays: −5.0	

✎ *Note that at step 5 the variable* `n` *of the void method contains the value 5.0 but when the flow of execution returns to the main code at step 6, the variable* `a` *of the main code still contains the value −5.0. Actually, the value of variable* `a` *of the main code had never changed!*

36.8 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) There are two categories of subprograms that return a value in Java.
2) The variables that are used to pass values to a method are called arguments.
3) The method `trim()` is a user-defined method.

4) Every call to a user-defined method is made in the same way as a call to the built-in methods of Java.
5) There can be as many arguments as you wish in a method's formal argument list.
6) In a method, the formal argument list must contain at least one argument.
7) In a method, the formal argument list is optional.
8) A method cannot return an array.
9) The following statement is a valid Java statement.

```
return x + 1;
```

10) A formal argument can be an expression.
11) An actual argument can be an expression.
12) A method can have no arguments in the actual argument list.
13) The next statement calls the method `cubeRoot()` three times.

```
cb = cubeRoot(x) + cubeRoot(x) / 2 + cubeRoot(x) / 3;
```

14) The following code fragment displays exactly the same value as the statement `System.out.println( cubeRoot(x) + 5);`
cb = **cubeRoot**(x); y = cb + 5;
System.out.**println**(y);
15) A method must always include a `return` statement whereas a void method mustn't.
16) The name `play-the-guitar` can be a valid method name.
17) In Java, you cannot place your methods below your main code.
18) When the main code calls a method, the execution of the statements of the main code is interrupted.
19) In general, it is possible for a function to return no values to the caller.
20) The method `indexOf()` is a built-in method of Java.
21) The following code fragment displays the value 0.5.

```
static double divide(double b, double a) {
   return a / b; }
public static void main(String[] args) throws Exception {
   double a = 10; double b = 5; System.out.println(divide(a, b)); }
```

22) In computer science, a subprogram that returns no result is known as a void function.
23) In Java, you can call a void method by writing its name followed by an opening and closing parenthesis.
24) In a void method call made in the main code, the variables used within the actual argument list must be variables from the main code.
25) In a void method call, only variables can be used within the actual argument list.
26) In a void method, all formal arguments must have different names.
27) A void method must always include at least one argument in its formal argument list.
28) There is a one-to-one correspondence between the formal and the actual arguments.
29) You can call a void method within a statement.
30) When the flow of execution reaches the end of a void method, the flow of execution continues from where it was before calling the void method.
31) A void method returns no values to the caller.
32) It is possible for a void method to accept no values from the caller.
33) A call to a void method is made differently from a call to a method.
34) In the following Java program, the first statement that executes is the statement `System.out.println( "Hello Aphrodite!")`.

```
static void message() {
   System.out.println("Hello Aphrodite!"); }
public static void main(String[] args) throws Exception {
   System.out.println("Hi there!"); message(); }
```

36.9 Review Exercises

Complete the following exercises.

1) The following method contains some errors. Can you spot them?

```
static int findMax(int a int b) if (a > b) {
      maximum = a;
   }
   else {
      maximum = b;
```

```
    }
}
```

2) Create a trace table to determine the values of the variables in each step of the following Java program.

```
static int sumDigits(int a) {
    int d1, d2;
    d1 = a % 10;
    d2 = (int)(a / 10);
    return d1 + d2; }
public static void main(String[] args) throws Exception {
    int s, i;
    s = 0;
    for (i = 25; i <= 27; i++) {
        s += sumDigits(i);
    }
    System.out.println(s); }
```

3) Create a trace table to determine the values of the variables in each step of the following Java program.

```
static int sss(int a) {
    int k, total;
    total = 0;
    for (k = 1; k <= a; k++) {
        total += k;
    }
    return total; }
public static void main(String[] args) throws Exception {
    int i, s;
    i = 1;
    s = 0;
    while (i < 6) {
        if (i % 2 == 1) {
            s += 1;
        }
        else {
            s += sss(i);
        }
        i++;
    }
    System.out.println(s); }
```

4) Create a trace table to determine the values of the variables in each step of the following Java program when the value 12 is entered.

```
static int customDiv(int b, int d) {
```

```
   return (int)((b + d) / 2); }
public static void main(String[] args) throws Exception {
   int k, m, a, x;
   k = Integer.parseInt(cin.nextLine()); m = 2;
   a = 1;
   while (a < 6) {
      if (k % m != 0) {
         x = customDiv(a, m);
      }
      else {
         x = a + m + customDiv(m, a);
      }
      System.out.println(m + " " + a + " " + x);
      a += 2;
      m++;
   }
}
```

5) Create a trace table to determine the values of the variables in each step of the following Java program when the values 3, 7, 9, 2, and 4 are entered.

```
static void display(int a) {
   if (a % 2 == 0) {
      System.out.println(a + " is even");
   }
   else {
      System.out.println(a + " is odd");
   }
}
public static void main(String[] args) throws Exception {
   int i, x;
   for (i = 1; i <= 5; i++) {
      x = Integer.parseInt(cin.nextLine());
      display(x);
   }
}
```

6) Create a trace table to determine the values of the variables in each step of the following Java program.

```
static void division(int a, int b) {
   b = (int)(b / a); System.out.println(a * b); }
public static void main(String[] args) throws Exception {
   int x, y;
   x = 20;
```

```
   y = 30;
   while (x % y < 30) {
      division(y, x);
      x = 4 * y;
      y++;
   }
}
```

7) Create a trace table to determine the values of the variables in each step of the following Java program when the values 2, 3, and 4 are entered.

```
static void calculate(int n) {
   int j; double s;
   s = 0;
   for (j = 2; j <= 2 * n; j += 2) {
      s = s + Math.pow(j, 2);
   }
   System.out.println(s); }
public static void main(String[] args) throws Exception {
   int i, m;
   for (i = 1; i <= 3; i++) {
      m = Integer.parseInt(cin.nextLine());
      calculate(m);
   }
}
```

8) Write a subprogram that accepts three integers through its formal argument list and then returns their sum.

9) Write a subprogram that accepts four numbers through its formal argument list and then returns their average.

10) Write a subprogram that accepts three numbers through its formal argument list and then returns the greatest value.

11) Write a subprogram that accepts five numbers through its formal argument list and then displays the greatest value.

12) Write a subprogram named `myRound` that accepts a real through its formal argument list and returns it rounded to two decimal places. Try not to use the `Math.round()` method of Java.

13) Do the following:
    i) Write a subprogram named `findMin` that accepts two numbers through its formal argument list and returns the lowest one.

ii) Using the subprogram cited above, write a Java program that prompts the user to enter four numbers and then displays the lowest one.

14) Do the following:

i) Write a subprogram named `KelvinToFahrenheit` that accepts a temperature in degrees Kelvin through its formal argument list and returns its degrees Fahrenheit equivalent.

ii) Write a subprogram named `KelvinToCelsius` that accepts a temperature in degrees Kelvin through its formal argument list and returns its degrees Celsius equivalent.

iii) Using the subprograms cited above, write a Java program that prompts the user to enter a temperature in degrees Kelvin and then displays its degrees Fahrenheit and its degrees Celsius equivalent.

It is given that Fahrenheit = 1.8 · Kelvin − 459.67

and

$$Celsius = Kelvin - 273.15$$

15) The Body Mass Index (BMI) is often used to determine whether a person is overweight or underweight for their height. The formula used to calculate the BMI is $BMI = \frac{weight \cdot 703}{height^2}$

Do the following:

i) Write a subprogram named `bmi` that accepts a weight and a height through its formal argument list and then returns an action (a message) according to the following table.

BMI	Action
BMI < 16	You must add weight.
16 ≤ BMI < 18.5	You should add some weight.
18.5 ≤ BMI < 25	Maintain your weight.
25 ≤ BMI < 30	You should lose some weight.
30 ≤ BMI	You must lose weight.

ii) Using the subprogram cited above, write a Java program that prompts the user to enter their weight (in pounds), age (in years),

and height (in inches), and then displays the corresponding message. Using a loop control structure, the program must also validate data input and display an error message when the user enters a) any negative value for weight b) any value less than 18 for age c) any negative value for height.

15) Do the following: i) Write a subprogram named `numOfDays` that accepts a year and a month (1 - 12) through its formal argument list and then displays the number of days in that month. Take special care when a year is a leap year; that is, a year in which February has 29 instead of 28 days.

Hint: A year is a leap year when it is exactly divisible by 4 and not by 100, or when it is exactly divisible by 400.

ii) Using the subprogram cited above, write a Java program that prompts the user to enter a year and then displays the number of the days in each month of that year.

17) Do the following: i) Write a subprogram named `numOfDays` that accepts a year and a month (1 - 12) through its formal argument list and then returns the number of days in that month. Take special care when a year is a leap year, as you did in the previous exercise.

ii) Using the subprogram cited above, write a Java program that prompts the user to enter a year and two months (1 - 12). The program must then calculate and display the total number of days that occur between the first day of the first month, and the last day of the second month.

18) Do the following: i) Write a subprogram named `displayMenu` that displays the following menu.

1) Convert meters to miles 2) Convert miles to meters 3) Exit

ii) Write a subprogram named `metersToMiles` that accepts a value in meters through its formal argument list and then displays the message “XX meters equals YY miles” where XX and YY must be replaced by actual values.

iii) Write a subprogram named `milesToMeters` that accepts a value in miles through its formal argument list and then displays the message “YY miles equals XX meters” where XX and YY must be replaced by actual values.

iv) Using the subprograms cited above, write a Java program that displays the previously mentioned menu and prompts the user to enter a choice (of 1, 2, or 3) and a distance. The program must then calculate and display the required value. The process must repeat as many times as the user wishes.

It is given that 1 mile = 1609.344 meters.

19) The LAV Cell Phone Company charges customers a basic rate of $10 per month, and additional rates are charged based on the total number of seconds a customer talks on their cell phone within the month. Use the rates shown in the following table.

Number of Seconds a Customer Talks on their Cell Phone	Additional Rates (in USD per second)
1 - 600	Free of charge
601 - 1200	$0.01
1201 and above	$0.02

Do the following: i) Write a subprogram named `amountToPay` that accepts a number in seconds through its formal argument list and then displays the total amount to pay. Please note that the rates are progressive. Moreover, federal, state, and local taxes add a total of 11% to each bill ii) Using the subprogram cited above, write a Java program that prompts the user to enter the number of seconds they talk on the cell phone and then displays the total amount to pay.

Chapter 37
Tips and Tricks with Subprograms

37.1 Can Two Subprograms use Variables of the Same Name?

Each subprogram uses its own memory space to hold the values of its variables. Even the main code has its own memory space! This means that you can have a variable named `test` in main code, another variable named `test` in a subprogram, and yet another variable named `test` in another subprogram. Pay attention! Those three variables are three completely different variables, in different memory locations, and they can hold completely different values.

As you can see in the program that follows, there are three variables named `test` in three different memory locations and each one of them holds a completely different value. The trace table below can help you understand what really goes on.

Project_37.1

```
static void f1() {
   String test;
   test = "Testing!"; System.out.println(test); }
static void f2(int test) {
   System.out.println(test); }
//Main code starts here public static void main(String[] args) throws Exception {
   int test;
   test = 5;
   System.out.println(test); f1(); f2(10); System.out.println(test); }
```

The trace table is shown here.

Step	Statement	Notes	Main Code	void Method `f1()`	void Method `f2()`
			test	test	test
1	`test = 5`		**5**		
2	`.println(test)`	It displays: 5	5		
3	`f1()`	`f1()` is called		?	
4	`test = "Testing!"`			**Testing!**	
5	`.println(test)`	It displays: Testing!		Testing!	
6	`f2(10)`	`f2()` is called			**10**
7	`.println(test)`	It displays: 10			10
8	`.println(test)`	It displays: 5	5		

✎ *Note that variables used in a subprogram "live" as long as the subprogram is being executed. This means that before calling the subprogram, none of its variables (including those in the formal argument list) exists in main memory (RAM). They are all defined in the main memory when the subprogram is called, and they are all removed from the main memory when*

the subprogram finishes and the flow of execution returns to the caller. The only variables that "live" forever, or at least for as long as the Java program is being executed, are the variables of the main code and the global variables! You will learn more about global variables in Section 37.6.

37.2 Can a Subprogram Call Another Subprogram?

Up to this point, you might have gotten the impression that only the main code can call a subprogram. However, this is not true! A subprogram can call any other subprogram which in turn can call another subprogram, and so on. You can make whichever combination you wish. For example, you can write a method that calls a void method, a void method that calls a method, a method that calls another method, or even a method that calls one of the built-in methods of Java.

The next example presents exactly this situation. The main code calls the void method `displaySum()`, which in turn calls the method `add()`.

Project_37.2

```
static int add(int number1, int number2) {
   int result;
   result = number1 + number2; return result; }
static void displaySum(int num1, int num2) {
   System.out.println(add(num1, num2)); }
public static void main(String[] args) throws Exception {
   int a, b;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine());
   displaySum(a, b); }
```

✎ *When the flow of execution reaches the* `return` *statement of the method* `add()`*, it returns to its caller, that is to the void method* `displaySum()`*. Then, when the flow of execution reaches the end of the void method* `displaySum()`*, it returns to its caller, that is, to the main code.*

✎ *Note that there is no restriction on the order in which the two subprograms should be written. It would have been exactly the same if the void method* `displaySum()` *had been written before the method* `add()`*.*

37.3 Passing Arguments by Value and by Reference

In Java, variables are passed to subprograms *by value*. This means that if the value of an argument is changed within the subprogram, it does not get changed outside of it. Take a look at the following example.

Project_37.3a

```
static void f1(int b) {
b++; //This is a variable of void method f1() System.out.println(b);
                        //It displays: 11
                        }
    public static void main(String[] args) throws Exception {
                        int a;
        a = 10; //This is a variable of the main code f1(a);
              System.out.println(a); //It displays: 10
                        }
```

The value 10 of variable `a` is passed to void method `f1()` through argument `b`. However, although the content of variable `b` is altered within the void method, when the flow of execution returns to the main code this change does not affect the value of variable `a`.

In the previous example, the main code and the void method are using two variables with different names. Yet, the same would have happened if, for instance, both the main code and the void method had used two variables of the same name. The next example operates exactly the same way and displays exactly the same results as the previous example did.

```
Project_37.3b
static void f1(int a) {
a++; //This is a variable of void method f1() System.out.println(a);
//It displays: 11
}
public static void main(String[] args) throws Exception {
int a;
a = 10; //This is a variable of the main code f1(a);
System.out.println(a); //It displays: 10
}
```

Passing an array to a subprogram as an argument is as easy as passing a simple variable. The next example passes array `a` to the void method `display()`, and the latter displays the array.

```
Project_37.3c
static final int ELEMENTS = 10;
static void display(int b[]) {
int i;
for (i = 0; i <= ELEMENTS - 1; i++) {
System.out.print(b[i] + "\t");
}
}
public static void main(String[] args) throws Exception {
int i;
int[] a = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
a[i] = Integer.parseInt(cin.nextLine());
}
display(a); }
```

Contrary to variables, data structures in Java are, by default, passed *by reference*. This means that if you pass, for example, an array to a subprogram, and that subprogram changes the value of one or more elements of the array, these changes are also reflected outside the subprogram. Take a look at the following example.

```
Project_37.3d
static void f1(int[] x) {
x[0]++;
System.out.println(x[0]); //It displays: 6
}
public static void main(String[] args) throws Exception {
int[] y = {5, 10, 15, 20};
```

```
        System.out.println(y[0]); //It displays: 5
    f1(y); System.out.println(y[0]); //It displays: 6
                        }
```

✎ *Passing an array to a subprogram passes a reference to the array, not a copy of the array, meaning that* `y` *and* `x` *are actually aliases of the same array. Only one copy of the array exists in the main memory (RAM). If a subprogram changes the value of an element, this change is also reflected in the main program.*

So, as you have probably realized, passing arrays by reference can provide an indirect way for a subprogram to "return" more than one value. In the next example, the method `myDivmod()` divides variable `a` by variable `b` and finds their integer quotient and their integer remainder. If all goes well, it returns `true`; otherwise, it returns `false`. Moreover, through the array `results`, the method also indirectly returns the calculated quotient and the calculated remainder.

```
                    🗀 Project_37.3e
    static boolean divMod(int a, int b, int[] results) {
                boolean returnValue = true;
                    if (b == 0) {
                returnValue = false;
                        }
                        else {
                results[0] = (int)(a / b);
                    results[1] = a % b;
                        }
                return returnValue; }
    public static void main(String[] args) throws Exception {
        int val1, val2; boolean ret; int[] res = new int[2];
          val1 = Integer.parseInt(cin.nextLine()); val2 =
Integer.parseInt(cin.nextLine()); ret = divMod(val1, val2, res); if
                            (ret) {
            System.out.println(res[0] + ", " + res[1]);
                            }
                            else {
        System.out.println("Sorry, wrong values entered!");
                            }
                        }
```

✎ *A very good tactic regarding the arguments in the formal argument list is to have all of those being passed by value written before those being passed by reference.*

37.4 Returning an Array

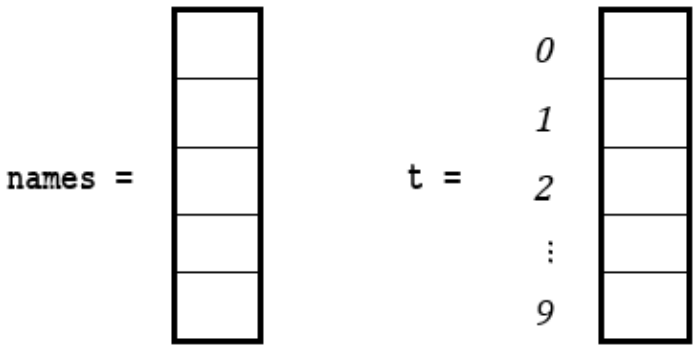
In the next example, the Java program must find the three lowest values of array `t`. To do so, the program calls and passes the array to the void method `getArray()` through its formal argument `x`, which in turn sorts array `x` using the insertion sort algorithm. When the flow of execution returns to the main code, array `t` is also sorted. This happens because, as already

stated, arrays in Java are passed by reference. So what the main code finally does is just display the values of the first three elements of the array.

```
Project_37.4a
static final int ELEMENTS = 10;
static void getArray(int[] x) {  [More…]
int m, n, element;
for (m = 1; m <= ELEMENTS - 1; m++) {
element = x[m];
n = m;
while (n > 0 && x[n - 1] > element) {
x[n] = x[n - 1];
n--;
}
x[n] = element;
}
}
public static void main(String[] args) throws Exception {
int i;
int[] t = {75, 73, 78, 70, 71, 74, 72, 69, 79, 77};
getArray(t);
System.out.println("Three lowest values are: ");
System.out.println(t[0] + " " + t[1] + " " + t[2]);
//In this step, array t is sorted for (i = 0; i <= ELEMENTS - 1;
i++) {
System.out.println(t[i]);
}
}
```

✎ *Since the array* `t` *of the main code is passed to the void method by reference, only one copy of the array exists in the main memory (RAM), meaning that* `t` *and* `x` *are actually aliases of the same array. When the flow of execution returns to the main code, the array* `t` *is also sorted.*

However, there are many times when passing an array by reference can be completely disastrous. Suppose you have the following two arrays. Array `names` contains the names of 10 cities, and array `t` contains their corresponding temperatures recorded at a specific hour on a specific day.



Now, suppose that for array `t` you wish to display the three lowest temperatures. If you call void method `getArray()` of the previous Java program, you have a problem. Although the three lowest temperatures can be displayed as required, the array `t` becomes sorted; therefore, the one-to-one correspondence between its elements and the elements of array `names` is lost forever!

One possible solution would be to write a method in which the array is copied to an auxiliary array and the method would return a smaller array that contains only the three lowest values. The proposed solution is shown here.

```
Project_37.4b
static final int ELEMENTS = 10;
static int[] getArray(int[] x) {
int m, n, element;
//Copy array x to array auxX
int[] auxX = new int[ELEMENTS]; for (m = 0; m <= ELEMENTS - 1; m++)
{
auxX[m] = x[m];
}
//and sort array auxX
for (m = 1; m <= ELEMENTS - 1; m++) {
element = auxX[m];
n = m;
while (n > 0 && auxX[n - 1] > element) {
auxX[n] = auxX[n - 1];
n--;
}
auxX[n] = element;
}
int[] retArray = {auxX[0], auxX[1], auxX[2]}; return retArray; }
public static void main(String[] args) throws Exception {
int i;
int[] t = {75, 73, 78, 70, 71, 74, 72, 69, 79, 77};
int[] low = getArray(t);
System.out.println("Three lowest values are: ");
System.out.println(low[0] + " " + low[1] + " " + low[2]);
//In this step, array t is NOT sorted for (i = 0; i <= ELEMENTS - 1;
i++) {
System.out.println(t[i]);
}
}
```

✎ *Note that you cannot use a statement such as* `int[] auxX = x` *to copy the elements of array* `x` *to* `auxX`*. This statement just creates two aliases of the same array. This is why a for-loop is used in the previous example to copy the elements of array* `x` *to the array* `auxX`*.*

37.5 Overloading Methods

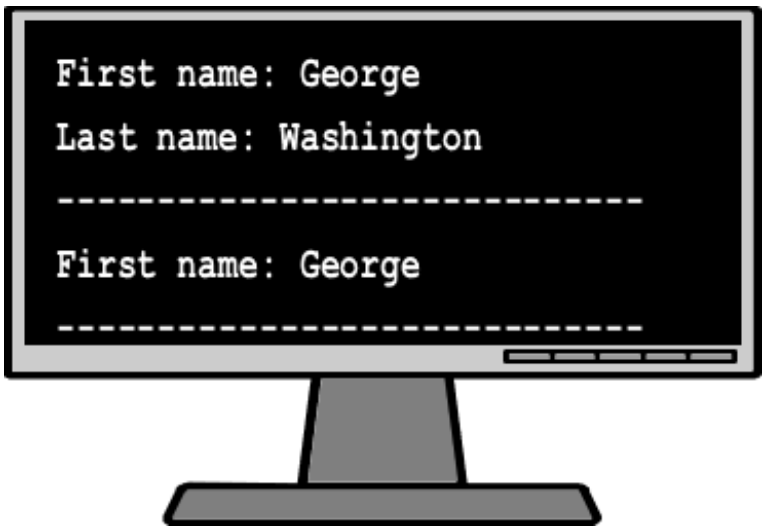
Method overloading is a feature that allows you to have two or more methods with the same name, as long as their formal argument lists are different. This means that you can perform overloading as long as: ► the number of arguments in each formal argument list is different; or ► the data type of the arguments in each formal argument list is different; or ► the sequence of the data type of the arguments in each formal argument list is different.

To better understand all these concepts, let's try to analyze them using one example for each.

Different number of arguments in each formal argument list In the following example, there are two methods `display()`, but they have different numbers of formal arguments. The first method accepts one string through its formal argument list, while the second accepts two strings.

```
Project_37.5a
    static void display(String firstName) {
        System.out.println("First name: " + firstName);
        System.out.println("----------------------------"); }
    static void display(String firstName, String lastName) {
        System.out.println("First name: " + firstName);
System.out.println("Last name: " + lastName); System.out.println("--
                ----------------------------"); }
    public static void main(String[] args) throws Exception {
      display("George", "Washington"); //This calls the second method
        display("George"); //This calls the first method }
```

The output result is shown here.

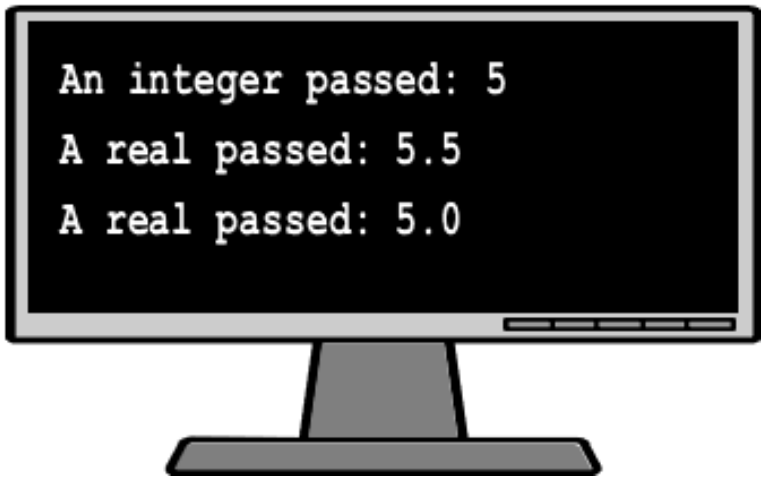


Different data type of arguments in each formal argument list In the following example, there are two methods `myAbs()`, but their formal arguments are of different types. The first method accepts an integer, while the second one accepts a real (a double) through its formal argument list.

```
Project_37.5b
static int myAbs(int a) {
        if (a < 0) {
            a *= -1;
```

```
                                    }
        System.out.print("An integer passed: "); return a; }
                static double myAbs(double a) {
                            if (a < 0) {
                                a *= -1;
                            }
          System.out.print("A real passed: "); return a; }
      public static void main(String[] args) throws Exception {
      System.out.println(myAbs(-5)); //This calls the first method
    System.out.println(myAbs(-5.5)); //This calls the second method
   System.out.println(myAbs(-5.0)); //This calls the second method }
```

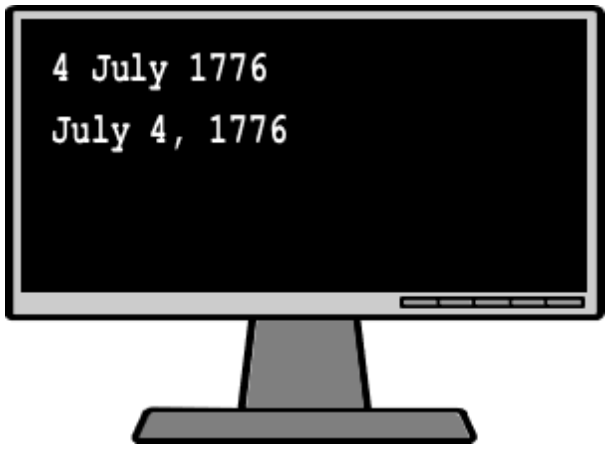
The output result is as follows.



Different sequence of the data types of arguments in each formal argument list Last but not least, in the following example, there are two methods `displayDate()`, but the sequence of the data types of the arguments in each formal argument list is different. The first one method accepts three arguments in the order string–integer–integer, while the second one accepts three arguments in the order integer–string–integer.

```
                        Project_37.5c
    static void displayDate(String month, int day, int year) {
       System.out.println(month + " " + day + ", " + year); }
    static void displayDate(int day, String month, int year) {
        System.out.println(day + " " + month + " " + year); }
     public static void main(String[] args) throws Exception {
     displayDate(4, "July", 1776); //This calls the second method
    displayDate("July", 4, 1776); //This calls the first method }
```

The output result is shown here.



37.6 The Scope of a Variable

The *scope of a variable* refers to the range of effect of that variable. In Java, a variable can have a *local* or *global scope*. A variable declared within a subprogram has a local scope and can be accessed only from within that subprogram. On the other hand, a variable declared outside of a subprogram has a global scope and can be accessed from within **any** subprogram, as well as from the main code.

Let's see some examples. The next example declares a global variable `test`. The value of this global variable, though, is accessed and displayed within the void method.

```
Project_37.6a
static int test; //Declare test as global
      static void displayValue() {
System.out.println(test); //It displays: 10
                }
public static void main(String[] args) throws Exception {
   test = 10; //This is a global variable displayValue();
      System.out.println(test); //It displays: 10
                }
```

✎ *To declare a global variable you must prepend the Java keyword* `static`.

✎ *Global variables must be declared outside of the main method. Most programmers prefer to have them all on the top for better observation.*

Be careful though! If the value of a global variable is altered within a subprogram, this change is also reflected outside of the subprogram. In the next example, the void method `displayValue()` increases the value of global variable `test` to 11, and when the flow of execution returns to the main code, a value of 11 is displayed.

```
Project_37.6b
static int test; //Declare test as global
      static void displayValue() {
                test++;
System.out.println(test); //It displays: 11
                }
```

```
public static void main(String[] args) throws Exception {
                        test = 10;
        System.out.println(test); //It displays: 10
 displayValue(); System.out.println(test); //It displays: 11
                        }
```

The next program declares a global variable `test`, a local variable `test` within the void method `displayValueA()`, and another local variable `test` within the void method `displayValueB()`. Keep in mind that the global variable `test` and the two local variables `test` are three different variables! Furthermore, the third method `displayValueC()` uses and alters the value of the global variable `test`. This is because there isn't any local variable `test` declared within this method.

```
                    Project_37.6c
          static int test; //Global variable test
               static void displayValueA() {
                int test; //Local variable test
                           test = 7;
          System.out.println(test); //It displays: 7
                           }
               static void displayValueB() {
                int test; //Local variable test
                           test = 9;
          System.out.println(test); //It displays: 9
                           }
               static void displayValueC() {
         //Use the value of the global variable test
         System.out.println(test); //It displays: 10
 test++; //Increase the value of the global variable test }
public static void main(String[] args) throws Exception {
        test = 10; //This is the global variable test
         System.out.println(test); //It displays: 10
displayValueA(); System.out.println(test); //It displays: 10
displayValueB(); System.out.println(test); //It displays: 10
displayValueC(); System.out.println(test); //It displays: 11
                        }
```

✎ *You can have variables of local scope of the same name within different subprograms, because they are recognized only by the subprogram in which they are declared.*

37.7 Converting Parts of Code into Subprograms

Writing large programs without subdividing them into smaller subprograms results in a code that cannot be easily understood or maintained. Suppose you have a large program and you wish to subdivide it into smaller subprograms. The next program is an example explaining the steps that must be followed. The parts of the program marked with a dashed rectangle must be converted into subprograms.

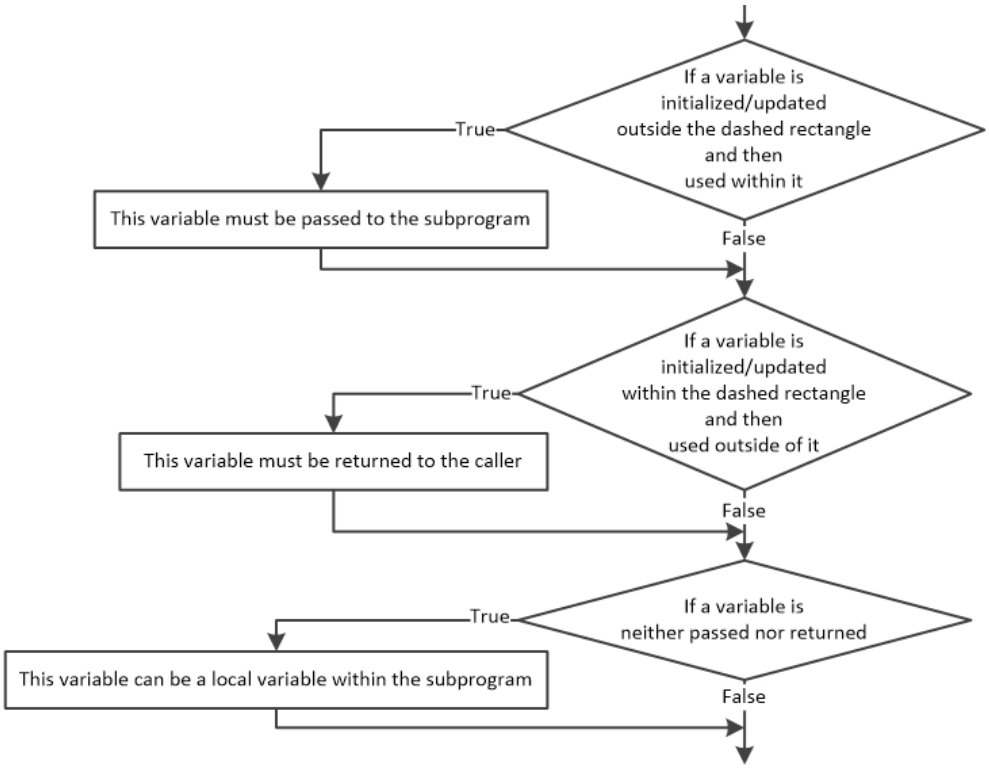
```
                    Project_37.7a
```

```
public static void main(String[] args) throws Exception {
int totalYes, femaleNo, i; String temp1, gender, temp2, answer;
totalYes = 0; femaleNo = 0; for (i = 1; i <= 100; i++) {
                    do {
System.out.print("Enter gender for citizen No " + i + ": ");
                    temp1 = cin.nextLine();
                  gender = temp1.toLowerCase();
} while (!gender.equals("male") && !gender.equals("female") &&
                  !gender.equals("other"));
                    do {
System.out.print("Do you go jogging in the afternoon? ");
                    temp2 = cin.nextLine();
                  answer = temp2.toLowerCase();
} while (!answer.equals("yes") && !answer.equals("no") &&
                !answer.equals("sometimes"));
                  if (answer.equals("yes")) {
                          totalYes++;
                             }
if (gender.equals("female") && answer.equals("no")) {
                          femaleNo++;
                             }
                           }
System.out.println("Total positive answers: " + totalYes);
System.out.println("Women's negative answers: " + femaleNo);
                           }
```

To convert parts of this program into subprograms you must: ► decide, for each dashed rectangle, whether to use a method or a void method. This depends on whether or not, the subprogram will return a result.

► determine which variables exist in each dashed rectangle and their roles in that dashed rectangle.

The flowchart that follows can help you decide what to do with each variable, whether it must be passed to the subprogram and/or returned from the subprogram, or if it must just be a local variable within the subprogram.


If a variable is initialized/updated outside the dashed rectangle and then used within it
True
This variable must be passed to the subprogram
False
If a variable is initialized/updated within the dashed rectangle and then used outside of it
True
This variable must be returned to the caller
False
If a variable is neither passed nor returned
True
This variable can be a local variable within the subprogram
False

So, with the help of this flowchart, let's deal with each dashed rectangle one by one! The parts that are **not** marked with a dashed rectangle will comprise the main code.

First part

In the first dashed rectangle, there are three variables: `i`, `temp1`, and `gender`. However, not all of them must be included in the formal argument list of the subprogram that will replace the dashed rectangle. Let's find out why!

- Variable `i`: is initialized/updated outside the dashed rectangle; thus, it **must** be passed to the subprogram is not updated within the dashed rectangle; thus, it should **not** be returned to the caller Variable `temp1`: is not initialized/updated outside of the dashed rectangle; thus, it should **not** be passed to the subprogram is initialized within the dashed rectangle but its value is not used outside of it; thus, it should **not** be returned to the caller According to the flowchart, since variable `temp1` should neither be passed nor returned, this variable can just be a **local variable** within the subprogram.
- Variable `gender`: is not initialized/updated outside of the dashed rectangle; thus, it should **not** be passed to the subprogram is initialized within the dashed rectangle and then its value is used outside of it; thus, it **must** be returned to the caller Therefore, since one value must be returned to the main code, a **method** can be used as shown here.

```
//First part static String getGender(int i) {
   String gender, temp1;
   do {
      System.out.print("Enter gender for citizen No " + i + ": ");
      temp1 = cin.nextLine();
      gender = temp1.toLowerCase();
   } while (!gender.equals("male") && !gender.equals("female") && !gender.equals("other"));
   return gender; }
```

Method's data type must match the data type of the value returned.

Second part

In the second dashed rectangle there are two variables, `temp2` and `answer`, but they do not both need to be included in the formal argument list of the subprogram that will replace the dashed rectangle. Let's find out why!

- Variable `temp2`: is not initialized/updated outside of the dashed rectangle; thus, it should **not** be passed to the subprogram is initialized/updated within the dashed rectangle but its value is not used outside of it; thus, it should **not** be returned to the caller According to the flowchart, since variable `temp2` should neither be passed nor returned, this variable can just be a **local variable** within the subprogram.
- Variable `answer`: is not initialized/updated outside of the dashed rectangle; thus, it should **not** be passed to the subprogram is initialized within the dashed rectangle and then its value is used outside of it; thus, it **must** be returned to the caller Therefore, since one value must be returned to the main code, a **method** can be used, as shown here.

```
//Second part static String getAnswer() {
   String temp2, answer;
   do {
      System.out.print("Do you go jogging in the afternoon? ");
      temp2 = cin.nextLine();
      answer = temp2.toLowerCase();
   } while (!answer.equals("yes") && !answer.equals("no") && !answer.equals("sometimes"));
   return answer; }
```

Third part

In the third dashed rectangle of the example, there are two variables: `totalYes` and `femaleNo`. Let's see what you should do with them.

- Both variables `totalYes` and `femaleNo`: are updated outside of the dashed rectangle; thus, they **must** be passed to the subprogram are not updated within the dashed rectangle; thus, they should **not** be returned to the caller Therefore, since no value should be returned to the main code, a **void method** can be used, as follows.

```
//Third part static void displayResults(int totalYes, int femaleNo) {
   System.out.println("Total positive answers: " + totalYes);
   System.out.println("Women's negative answers: " + femaleNo); }
```

The final program The final program, including the main code and all the subprograms cited above, is shown here.

```
Project_37.7b
//First part static String getGender(int i) {
String gender, temp1;
do {
System.out.print("Enter gender for citizen No " + i
+ ": ");
temp1 = cin.nextLine();
gender = temp1.toLowerCase();
} while (!gender.equals("male") &&
!gender.equals("female") && !gender.equals("other"));
```

```
                    return gender; }
        //Second part static String getAnswer() {
                    String temp2, answer;
                                do {
          System.out.print("Do you go jogging in the
                                afternoon? ");
                    temp2 = cin.nextLine();
                  answer = temp2.toLowerCase();
   } while (!answer.equals("yes") && !answer.equals("no")
             && !answer.equals("sometimes"));
                    return answer; }
//Third part static void displayResults(int totalYes, int
                    femaleNo) {
      System.out.println("Total positive answers: " +
      totalYes); System.out.println("Women's negative
                answers: " + femaleNo); }
public static void main(String[] args) throws Exception {
     int i, totalYes, femaleNo; String gender, answer;
   totalYes = 0; femaleNo = 0; for (i = 1; i <= 100; i++)
                                {
                   gender = getGender(i);
                    answer = getAnswer();
                 if (answer.equals("yes")) {
                            totalYes++;
                               }
     if (gender.equals("female") && answer.equals("no"))
                               {
                            femaleNo++;
                               }
                              }
           displayResults(totalYes, femaleNo); }
```

37.8 Recursion

Recursion is a programming technique in which a subprogram calls itself. This might initially seem like an endless loop, but of course this is not true; a subprogram that uses recursion must be written in a way that obviously satisfies the property of finiteness.

Imagine that the next Java program helps you find your way home. In this program, recursion occurs because the void method `find_your_way_home()` calls itself within the method.

```
static void find_your_way_home() {
   if (you_are_already_at_home) {
      stop_walking();
   }
   else {
      take_one_step_toward_home();
      find_your_way_home();
```

```
   }
}
public static void main(String[] args) throws Exception {
   find_your_way_home(); }
```

Now, let's try to analyze recursion through a real example. The next Java program calculates the factorial of 5 using recursion.

Project_37.8

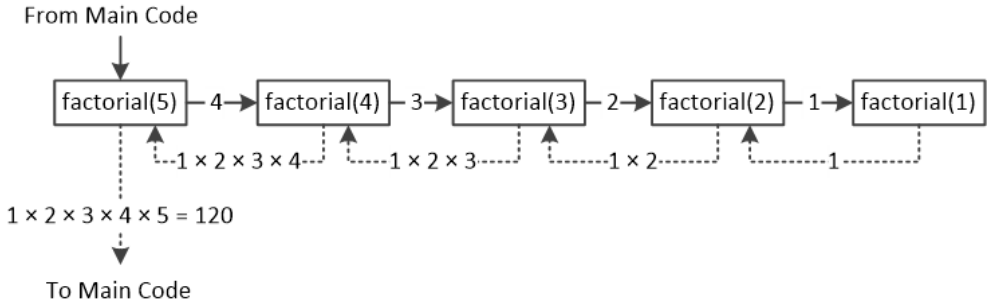
```
static int factorial(int value) {
   if (value == 1) {
      return 1;
   }
   else {
      return value * factorial(value - 1);
   }
}
public static void main(String[] args) throws Exception {
   System.out.println(factorial(5)); //It displays: 120
}
```

In mathematics, the factorial of a non-negative integer N is the product of all positive integers less than or equal to N. It is denoted by N! and the factorial of 0 is, by definition, equal to 1. For example, the factorial of 5 is 1 × 2 × 3 × 4 × 5 = 120.

Recursion occurs because the method `factorial()` *calls itself within the method.*

Note that there isn't any loop control structure!

You are probably confused right now. How on Earth is the product 1 × 2 × 3 × 4 × 5 calculated without using a loop control structure? The next diagram may help you understand. It shows the multiplication operations that are performed as method `factorial(5)` works its way backwards through the series of calls.



Let's see how this diagram works. The main code calls the method `factorial(5)`, which in turn calls the method `factorial(4)`, and the latter calls the method `factorial(3)`, and so on. The last call (`factorial(1)`) returns to its caller (`factorial(2)`) the value 1, which in turn returns to its caller (`factorial(3)`) the value 1 × 2 = 2, and so on. When the method `factorial(5)` returns from the topmost call, you have the final solution.

To avoid logic errors, all recursive subprograms must adhere to three important rules: 1) They must call themselves.

2) They must have a *base case*, which is the condition that “tells” the subprogram to stop recursions. The base case is usually a very small problem that can be solved directly. It is the solution to the "simplest" possible problem. In the method `factorial()` of the previous example, the base case is the factorial of 1. When `factorial(1)` is called, the Boolean expression `value == 1` evaluates to `true` and signals the end of the recursions.
3) They must change their state and move toward the base case. A change of state means that the subprogram alters some of its data. Usually, data are getting smaller and smaller in some way. In the function `factorial()` of the previous example, since the base case is the factorial of 1, the whole concept relies on the idea of moving toward that base case.

In conclusion, recursion helps you write more creative and more elegant programs, but keep in mind that it is not always the best option. The main disadvantage of recursion is that it is hard for a programmer to think through the logic, and therefore it is difficult to debug a code that contains a recursive subprogram. Furthermore, a recursive algorithm may prove worse than a non-recursive one because it may consume too much CPU time and/or too much main memory (RAM). So, there are times where it would be better to follow the KISS principle and, instead of using a recursion, solve the algorithm using loop control structures.

✎ *For you who don't know what the KISS principle is, it is an acronym for “Keep It Simple, Stupid”! It states that most systems work best if they are kept simple, avoiding any unnecessary complexity!*

37.9 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Each subprogram uses its own memory space to hold the values of its variables.
2) Variables used in a subprogram "live" as long as the subprogram is being executed.
3) The only variables that “live” for as long as the Java program is being executed are the variables of the main code and the global variables.
4) A subprogram can call the main code.
5) If an argument is passed by value and its value is changed within the subprogram, it does not get changed outside of it.
6) The name of an actual argument and the name of the corresponding formal argument must be the same.
7) The total number of actual arguments cannot be greater than the total number of formal arguments.
8) An expression cannot be passed to a subprogram.
9) By default, arrays in Java are passed by reference.
10) You can pass an array to a void method but the void method cannot return (directly or indirectly) an array to the caller.
11) A method can accept an array through its formal argument list.

12) In general, a void function can call any function.
13) In general, a function can call any void function.
14) Within a statement, a method can be called only once.
15) A void method can return a value through its formal argument list.
16) A subprogram can be called by another subprogram or by the main code.
17) Overloading is a feature in Java that allows you to have two or more arguments with the same name.
18) You can perform overloading as long as the number of arguments in the formal argument lists is equal.
19) Overloading is a feature in Java that allows you to have two or more methods of different names as long as they have exactly the same formal argument list.
20) The scope of a variable refers to the range of effect of that variable.
21) If the value of a global variable is altered within a subprogram, this change is reflected outside the subprogram as well.
22) You can have two variables of global scope of the same name.
23) Recursion is a programming technique in which a subprogram calls itself.
24) A recursive algorithm must have a base case.
25) Using recursion to solve a problem is not always the best option.

37.10 Review Exercises

Complete the following exercises.

1) Without using a trace table, can you find out what the next Java program displays?

```
static void f1() {
   int a = 22; }
static void f2() {
   int a = 33; }
public static void main(String[] args) throws Exception {
   int a;
   a = 5;
   f1(); f2();
   System.out.println(a); }
```

2) Without using a trace table, can you find out what the next Java program displays?

```
static int f1(int number1) {
   return 2 * number1; }
static int f2(int number1, int number2) {
   return f1(number1) + f1(number2); }
public static void main(String[] args) throws Exception {
   int a, b;
   a = 3;
   b = 4;
   System.out.println(f2(a, b)); }
```

3) Without using a trace table, can you find out what the next Java program displays?

```
static int f1(int number1) {
   return 2 * number1; }
static int f2(int number1, int number2) {
   number1 = f1(number1); number2 = f1(number2);
   return number1 + number2; }
public static void main(String[] args) throws Exception {
   int a, b;
   a = 2;
   b = 5;
   System.out.println(f2(a, b)); }
```

4) Without using a trace table, can you find out what the next Java program displays?

```
static void display(String str) {
   str = str.replace("a", "e"); System.out.print(str); }
static void display() {
   System.out.print("hello"); }
public static void main(String[] args) throws Exception {
   display("hello"); display(); display("hallo"); }
```

5) Without using a trace table, can you find out what the next Java program displays?

```
static int a, b;
static void f1() {
   a = a + b;
}
public static void main(String[] args) throws Exception {
   a = 10;
   b = 5;
   f1(); b--;
   System.out.println(a); }
```

6) Without using a trace table, can you find out what the next Java program displays?

```
static int a, b;
static void f1() {
   a = a + b;
   f2(); }
static void f2() {
   a = a + b;
}
public static void main(String[] args) throws Exception {
   a = 3;
   b = 4;
   f1(); System.out.println(a + " " + b); }
```

7) Without using a trace table, can you find out what the next Java program displays?

```
static int foo(int a[], int b) {
   int c = 0; for (int x : a) {
      if (x == b)
         c++;
   }
   return c; }
public static void main(String[] args) throws Exception {
   int[] a = {5, 9, 2, 5, 5}; System.out.println(foo(a, 5)); }
```

8) The following Java program is supposed to prompt the user to enter five integers into an array and then display, for each element, its number of digits and the integer itself. For example, if the user enters the values 35, 13565, 113, 278955, 9999, the program is supposed to display:
2 digits in number 35
5 digits in number 13565
3 digits in number 113
6 digits in number 278955
4 digits in number 9999

Unfortunately, the program displays

2 digits in number 0
5 digits in number 0
3 digits in number 0
6 digits in number 0
4 digits in number 0

Can you find out why?

```
static final int ELEMENTS = 5;
static int getNumOfDigits(int[] x, int index) {
   int count = 0;
   while (x[index] != 0) {
      count++;
      x[index] = (int)(x[index] / 10);
   }
   return count; }
public static void main(String[] args) throws Exception {
   int[] val = new int[ELEMENTS]; int i;
   for (i = 0; i <= ELEMENTS - 1; i++) {
      val[i] = Integer.parseInt(cin.nextLine());
   }
   for (i = 0; i <= ELEMENTS - 1; i++) {
      System.out.print(getNumOfDigits(val, i) + " digits in number " + val[i]);
   }
}
```

9) For the following Java program, convert the parts marked with a dashed rectangle into subprograms.

```
static final int STUDENTS = 10; static final int LESSONS = 5;
public static void main(String[] args) throws Exception {
   int i, j, m, n; double temp; String tempStr;
   String[] names = new String[STUDENTS]; int[][] grades = new int[STUDENTS][LESSONS];
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.print("Enter name No" + (i + 1) + ": ");
      names[i] = cin.nextLine();
      for (j = 0; j <= LESSONS - 1; j++) {
         System.out.print("Enter grade for lesson No" + (j + 1) + ": ");
         grades[i][j] = Integer.parseInt(cin.nextLine());
      }
   }
```

```
   double[] average = new double[STUDENTS]; for (i = 0; i <= STUDENTS - 1; i++) {
      average[i] = 0;
      for (j = 0; j <= LESSONS - 1; j++) {
         average[i] += grades[i][j];
      }
      average[i] /= LESSONS;
   }
```

```
   for (m = 1; m <= STUDENTS - 1; m++) {
      for (n = STUDENTS - 1; n >= m; n--) {
         if (average[n] > average[n - 1]) {
            temp = average[n];
            average[n] = average[n - 1];
            average[n - 1] = temp;
            tempStr = names[n];
            names[n] = names[n - 1];
            names[n - 1] = tempStr;
         }
         else if (average[n] == average[n - 1]) {
            if (names[n].compareTo(names[n - 1]) < 0) {
               tempStr = names[n];
               names[n] = names[n - 1];
               names[n - 1] = tempStr;
            }
         }
      }
   }
```

```
   for (i = 0; i <= STUDENTS - 1; i++) {
      System.out.println(names[i] + "\t" + average[i]);
   }
}
```

10) For the following Java program, convert the parts marked with a dashed rectangle into subprograms.

```
public static void main(String[] args) throws Exception {
   int i, middlePos, j; String message, messageClean, letter, leftLetter, rightLetter; boolean
   palindrome;
```

```
   System.out.print("Enter a message: "); message = cin.nextLine().toLowerCase();
   messageClean = ""; for (i = 0; i <= message.length() - 1; i++) {
      letter = "" + message.charAt(i);
      if (!letter.equals(" ") && !letter.equals(",") &&
            !letter.equals(".") && !letter.equals("?")) {
         messageClean += letter;
      }
   }
   middlePos = (int)((messageClean.length() - 1) / 2); j = messageClean.length() - 1;
   palindrome = true; for (i = 0; i <= middlePos; i++) {
      leftLetter = "" + messageClean.charAt(i);
      rightLetter = "" + messageClean.charAt(j);
      if (!leftLetter.equals(rightLetter)) {
         palindrome = false;
         break;
      }
      j--;
   }
   if (palindrome) {
      System.out.println("The message is palindrome");
   }
}
```

11) The next Java program finds the greatest value among four user-provided values. Rewrite the program without using subprograms.

```
static int myMax(int n, int m) {
   if (n > m) {
      m = n;
   }
   return m; }
public static void main(String[] args) throws Exception {
   int a, b, c, d, maximum;
   a = Integer.parseInt(cin.nextLine()); b = Integer.parseInt(cin.nextLine()); c =
   Integer.parseInt(cin.nextLine()); d = Integer.parseInt(cin.nextLine());
   maximum = a; maximum = myMax(b, maximum); maximum = myMax(c, maximum); maximum = myMax(d,
   maximum);
   System.out.println(maximum); }
```

12) Write a void method that accepts three numbers through its formal argument list and then returns their sum and their average.

    Hint: Use an array within the formal argument list to “return” the required values.

13) Write a subprogram named `myRound` that accepts a real (a float) and an integer through its formal argument list and then returns the real rounded to as many decimal places as the integer indicates. Moreover, if no value is passed for the integer, the subprogram must return the real rounded to two decimal places by default. Try not to use the `Math.round()` method of Java.

    Hint: Use overloading.

14) Do the following:
    i) Write a subprogram named `getInput` that prompts the user to enter an answer “yes” or “no” and then returns the value `true` or `false` correspondingly to the caller. Make the subprogram accept the answer in all possible forms such as “yes”, “YES”, “Yes”, “No”, “NO”, “nO”, and so on.

ii) Write a subprogram named `findArea` that accepts the base and the height of a parallelogram through its formal argument list and then returns its area.

iii) Using the subprograms cited above, write a Java program that prompts the user to enter the base and the height of a parallelogram and then calculates and displays its area. The program must iterate as many times as the user wishes. At the end of each calculation, the program must ask the user whether they wish to calculate the area of another parallelogram. If the answer is "yes" the program must repeat.

15) Do the following:
i) Write a subprogram named `getArrays` that prompts the user to enter the grades and the names of 100 students into the arrays `grades` and `names`, correspondingly. The two arrays must be returned to the caller.

ii) Write a subprogram named `getAverage` that accepts the array `grades` through its formal argument list and returns the average grade.

iii) Write a subprogram named `sortArrays` that accepts the arrays `grades` and `names` through its formal argument list and sorts the array `grades` in descending order using the insertion sort algorithm. The subprogram must preserve the one-to-one correspondence between the elements of the two arrays.

iv) Using the subprograms cited above, write a Java program that prompts the user to enter the grades and the names of 100 students and then displays all student names whose grade is less than the average grade, sorted by grade in descending order.

16) In a song contest, there is an artist who is scored by 10 judges. However, according to the rules of this contest, the total score is calculated after excluding the highest and lowest scores. Do the following:
i) Write a subprogram named `getArray` that prompts the user to enter the scores of the 10 judges into an array and then returns the array to the caller. Assume that each score is unique.

ii) Write a subprogram named `findMinMax` that accepts an array through its formal argument list and then returns the maximum and the minimum value.

iii) Using the subprograms cited above, write a Java program that prompts the user to enter the name of the artist and the score they get from each judge. The program must then display the message "Artist NN got XX points" where NN and XX must be replaced by actual values.

17) Do the following:
i) Write a recursive method named `sumRecursive` that accepts an integer through its formal argument list and then returns the sum of numbers from 1 to that integer.

ii) Using the subprogram cited above, write a Java program that lets the user enter a positive integer, and then displays the sum of numbers from 1 to that user-provided integer.

18) On a chessboard you must place grains of wheat on each square, such that one grain is placed on the first square, two on the second, four on the third, and so on (doubling the number of grains on each subsequent square). Do the following:
i) Write a recursive method named `woc` that accepts the index of a square and returns the number of grains of wheat that are on this square. Since a chessboard contains $8 \times 8 = 64$ squares, assume that the index is an integer between 1 and 64.

ii) Using the subprogram cited above, write a Java program that calculates and displays the total number of grains of wheat that are on the chessboard in the end.

19) The Fibonacci sequence is a series of numbers in the following sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

By definition, the first two terms are 0 and 1 and each subsequent term is the sum of the previous two.

Do the following:

i) Write a recursive method named `fib` that accepts an integer through its formal argument list and then returns the N^{th} term of the Fibonacci series.

ii) Using the subprogram cited above, write a Java program that lets the user enter a positive integer N and then displays the N^{th} term of the Fibonacci series.

20) The Tribonacci sequence is similar to the Fibonacci sequence but each term is the sum of the three preceding terms. Write a recursive method named `trib` that accepts an integer through its formal argument list and then returns the N^{th} term of the Tribonacci series.

21) Write a recursive method named `myPow` that accepts a real and an integer and then returns the result of the first number raised to the power of the second number, without using the built-in `Math.pow()` method of Java. Ensure that the function works correctly for both positive and negative exponent values.

22) Do the following:

i) Write a recursive method named `factorial` that accepts an integer through its formal argument list and then returns its factorial.

ii) Using the method cited above, write a recursive method named `myCos` that calculates and returns the cosine of *x* using the Taylor series, shown next.

$$cosx = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Hint: Keep in mind that *x* is in radians and $\frac{x^0}{0!} = 1$.

iii) Using the method `myCos()` cited above, write a Java program that calculates and displays the cosine of 45^o.

Hint: To verify the result, note that the cosine of 45^o is approximately 0.7071067811865475.

Chapter 38
More with Subprograms

38.1 Simple Exercises with Subprograms

Exercise 38.1-1 A Simple Currency Converter

Do the following: i) Write a subprogram named `displayMenu` *that displays the following menu.*

1) *Convert USD to Euro (EUR)*
2) *Convert Euro (EUR) to USD*
3) *Exit*

ii) Using the subprogram cited above, write a Java program that displays the previously mentioned menu and prompts the user to enter a choice (of 1, 2, or 3). If choice 1 or 2 is selected, the program must prompt the user to enter an amount of money and then it must calculate and display the corresponding converted value. The process must repeat as many times as the user wishes.

It is given that $1 = 0.94 EUR (€).

Solution

According to the "Ultimate" rule, the while-loop of the main code must be as follows, given in general form.

```
displayMenu(); choice = Integer.parseInt(cin.nextLine()); //Initialization of choice.
while (choice != 3) {

    Prompt the user to enter an amount of money,
    and then calculate and display the required
    value.

    displayMenu(); choice = Integer.parseInt(cin.nextLine()); //Update/alteration of
    choice }
```

The solution is as follows.

Project_38.1-1

```
static void displayMenu() {
    System.out.println("1. Convert USD to Euro (EUR)"); System.out.println("2. Convert
    Euro (EUR) to USD"); System.out.println("3. Exit"); System.out.println("----------
    ---------------"); System.out.print("Enter a choice: "); }
public static void main(String[] args) throws Exception {
    int choice; double amount;
    displayMenu(); choice = Integer.parseInt(cin.nextLine()); while (choice != 3) {
```

```
      System.out.print("Enter an amount: ");
      amount = Double.parseDouble(cin.nextLine());
      if (choice == 1) {
         System.out.println(amount + " USD = " + amount * 0.94 + " Euro");
      }
      else {
         System.out.println(amount + " Euro = " + amount / 0.94 + " USD");
      }
      displayMenu();
      choice = Integer.parseInt(cin.nextLine());
   }
}
```

Exercise 38.1-2 Finding the Average Values of Positive Integers

Do the following: i) Write a subprogram named `testInteger` *that accepts a number through its formal argument list and returns* `true` *when the passed number is an integer; it must return* `false` *otherwise.*

ii) Using the subprogram cited above, write a Java program that lets the user enter numeric values repeatedly until a real one is entered. In the end, the program must display the average value of positive integers entered.

Solution

To solve this exercise, a `while` statement will be used. According to the "Ultimate" rule discussed in Section 29.3, the pre-test loop structure that solves this problem should be as follows.

```
x = Double.parseDouble(cin.nextLine()); //Initialization of x while (testInteger(x))
{ //Boolean Expression dependent on x

A statement or block of statements

   x = Double.parseDouble(cin.nextLine()); //Update/alteration of x }
```

✎ *The statement* `while (testInteger(x))` *is equivalent to the statement* `while (testInteger(x) == true)`.

The final solution is presented next.

Project_38.1-2

```
static boolean testInteger(double number) {
   boolean returnValue = false;
   if (number == (int)(number)) {
```

```
      returnValue = true;
   }
   return returnValue; }
public static void main(String[] args) throws Exception {
   int count; double total, x;
   total = 0;
   count = 0;
   x = Double.parseDouble(cin.nextLine()); //Initialization of x while
   (testInteger(x)) { //Boolean Expression dependent on x
   if (x > 0) {
         total += x;
         count++;
      }
      x = Double.parseDouble(cin.nextLine()); //Update/alteration of x
   }
   if (count > 0) {
      System.out.println(total / count);
   }
}
```

✎ *Note the last single-alternative decision structure,* `if (count > 0)`*. It is necessary in order for the program to satisfy the property of definiteness. Think about it! If the user enters a real (float) right from the beginning, the variable* `count`*, in the end, will contain a value of zero.*

✎ *The following method can be used as an alternative to the previous one. It directly returns the result (*`true` *or* `false`*) of the Boolean expression* `number == (int)(number)`**.**

```
static boolean test_integer(double number) {
   return number == (int)(number); }
```

Exercise 38.1-3 Finding the Sum of Odd Positive Integers

Do the following: i) Write a subprogram named `testInteger` *that accepts a number through its formal argument list and returns* `true` *when the passed number is an integer; it must return* `false` *otherwise.*

ii) Write a subprogram named `testOdd` *that accepts a number through its formal argument list and returns* `true` *when the passed number is odd; it must return* `false` *otherwise.*

iii) Write a subprogram named `testPositive` *that accepts a number through its formal argument list and returns* `true` *when the passed number is positive; it must return* `false` *otherwise.*

iv) Using the subprograms cited above, write a Java program that lets the user enter numeric values repeatedly until a negative one is entered. In the end, the program must display the sum of odd positive integers entered.

Solution

This exercise is pretty much the same as the previous one. Each subprogram returns one value (which can be `true` or `false`). The solution is presented here.

Project_38.1-3

```
static boolean testInteger(double number) {
   return number == (int)(number); }
static boolean testOdd(double number) {
   return number % 2 != 0; }
static boolean testPositive(double number) {
   return number > 0; }
public static void main(String[] args) throws Exception {
   int total; double x;
   total = 0;
   x = Double.parseDouble(cin.nextLine()); while (testPositive(x)) {
      if (testInteger(x) && testOdd(x)) {
         total += (int)x;
      }
      x = Double.parseDouble(cin.nextLine());
   }
   System.out.println(total); }
```

✎ *The statement* `if (testInteger(x) && testOdd(x))` *is equivalent to the statement* `if (testInteger(x) == true && testOdd(x) == true)`

Exercise 38.1-4 Finding the Values of y

Write a Java program that finds and displays the value of y (if possible) in the following formula.

$$y = \begin{cases} \dfrac{3x}{x-5} + \dfrac{7-x}{2x}, & x \geq 1 \\ \dfrac{45-x}{x+2} + 3x, & x < 1 \end{cases}$$

For each part of the formula, write a subprogram that accepts `x` *through its formal argument list and then calculates and displays the result. The subprogram must display an error message when the calculation is not possible.*

Solution

Each subprogram must calculate and display the result of the corresponding formula or display an error message when the calculation is not possible. As these two subprograms return no result, they can both be written as void methods. The solution is shown here.

Project_38.1-4

```
static void formula1(double x) {
   double y;
   if (x == 5) { //No need to check for $x == 0 when $x >= 1
      System.out.println("Error! Division by zero");
   }
   else {
      y = 3 * x / (x - 5) + (7 - x) / (2 * x);
      System.out.println(y);
   }
}
static void formula2(double x) {
   double y;
   if (x == -2) {
      System.out.println("Error! Division by zero");
   }
   else {
      y = (45 - x) / (x + 2) + 3 * x;
      System.out.println(y);
   }
}
public static void main(String[] args) throws Exception {
   double x;
   System.out.print("Enter a value for x: "); x = Double.parseDouble(cin.nextLine());
   if (x >= 1) {
      formula1(x);
   }
   else {
      formula2(x);
   }
}
```

38.2 Exercises of a General Nature with Subprograms

Exercise 38.2-1 Validating Data Input Using a Subprogram

Do the following: i) Write a subprogram named `getAge` *that prompts the user to enter their age and returns it. Using a loop control structure, the subprogram must also validate data input and display an error message when the user enters any non-positive values.*

ii) Write a subprogram named `findMax` *that accepts an array through its formal argument list and returns the index position of the maximum value of the array.*

iii) Using the subprograms cited above, write a Java program that prompts the user to enter the first names, last names, and ages of 50 people into three arrays and then displays the name of the oldest person.

Solution

Since the subprogram `getAge()` returns one value, it can be written as a method. The same applies to subprogram `findMax()` because it also returns one value. The main code must prompt the user to enter the first names, the last names, and the ages of 50 people into arrays `firstNames`, `lastNames`, and `ages` respectively. Then, with the help of method `findMax()`, it can find the index position of the maximum value of array `ages`. The solution is shown here.

Project_38.2-1

```
static final int PEOPLE = 50;
static int getAge() {
   int age;
   System.out.print("Enter an age: "); age = Integer.parseInt(cin.nextLine()); while
   (age <= 0) {
      System.out.print("Error: Invalid age\nEnter a positive number: ");
      age = Integer.parseInt(cin.nextLine());
   }
   return age; }
static int findMax(int[] a) {
   int i, maximum, maxI;
   maximum = a[0];
   maxI = 0;
   for (i = 1; i <= PEOPLE - 1; i++) {
      if (a[i] > maximum) {
```

```
            maximum = a[i];
            maxI = i;
         }
      }
      return maxI; }
   public static void main(String[] args) throws Exception {
      int i, indexOfMax;
      String[] firstNames = new String[PEOPLE]; String[] lastNames = new String[PEOPLE];
      int[] ages = new int[PEOPLE]; for (i = 0; i <= PEOPLE - 1; i++) {
         System.out.print("Enter first name of person No " + (i + 1) + ": ");
         firstNames[i] = cin.nextLine();
         System.out.print("Enter last name of person No " + (i + 1) + ": ");
         lastNames[i] = cin.nextLine();
         ages[i] = getAge();
      }
      indexOfMax = findMax(ages);
      System.out.println("The oldest person is:");
      System.out.println(firstNames[indexOfMax] + lastNames[indexOfMax]);
      System.out.println("They are " + ages[indexOfMax] + " years old!"); }
```

Exercise 38.2-2 Sorting an Array Using a Subprogram

Do the following: i) Write a subprogram named `mySwap` *that accepts an array of strings through its formal argument list, as well as two indexes. The subprogram then swaps the values of the elements at the corresponding index positions.*

ii) Using the subprogram `mySwap()` *cited above, write a subprogram named* `mySort` *that accepts an array of string through its formal argument list and then sorts the array using the bubble sort algorithm. It must be able to sort in either ascending or descending order. To do this, include an addition Boolean argument within the formal argument list.*

iii) Write a subprogram named `displayArray` *that accepts an array through its formal argument list and then displays it.*

iv) Using the subprograms `mySort()` *and* `displayArray()` *cited above, write a Java program that prompts the user to enter the names of 20 people and then displays them twice: once sorted in ascending order, and once in descending order.*

Solution

As you can see in the Java program below, the void method `mySort()` uses an adapted version of the bubble sort algorithm. When the value `true` is passed to the argument `ascending`, the algorithm sorts array `a` in ascending order. When the value `false` is passed, the algorithm sorts array `a` in descending order.

Moreover, the void method `mySort()` calls the void method `mySwap()` every time a swap is required between the contents of two elements.

Project_38.2-2

```
static final int PEOPLE = 20;
static void mySwap(String[] a, int index1, int index2) {
   String temp;
   temp = a[index1]; a[index1] = a[index2]; a[index2] = temp; }
static void mySort(String[] a, boolean ascending) {
   int m, n;
   for (m = 1; m <= PEOPLE - 1; m++) {
      for (n = PEOPLE - 1; n >= m; n--) {
         if (ascending) {
            if (a[n].compareTo(a[n - 1]) < 0) {
               mySwap(a, n, n - 1);
            }
         }
         else {
            if (a[n].compareTo(a[n - 1]) > 0) {
               mySwap(a, n, n - 1);
            }
         }
      }
   }
}
static void displayArray(String[] a) {
   int i;
   for (i = 0; i <= PEOPLE - 1; i++) {
      System.out.println(a[i]);
   }
}
public static void main(String[] args) throws Exception {
   int i;
   String[] names = new String[PEOPLE]; for (i = 0; i <= PEOPLE - 1; i++) {
      System.out.print("Enter a name: ");
      names[i] = cin.nextLine();
   }
```

```
    mySort(names, true); //Sort names in ascending order displayArray(names); //and
    display them
    mySort(names, false); //Sort names in descending order displayArray(names); //and
    display them.
}
```

✎ *In Java, arrays are passed by reference. This is why there is no need to include a* `return` *statement in the subprograms* `mySwap()` *and* `mySort()`.

Exercise 38.2-3 Progressive Rates and Electricity Consumption

The LAV Electricity Company charges subscribers for their electricity consumption according to the following table (monthly rates for domestic accounts).

Kilowatt-hours (kWh)	USD per kWh
kWh ≤ 400	$0.08
401 ≤ kWh ≤ 1500	$0.22
1501 ≤ kWh ≤ 2000	$0.35
2001 ≤ kWh	$0.50

Do the following: i) Write a subprogram named `getConsumption` *that prompts the user to enter the total number of kWh consumed and then returns it. Using a loop control structure, the subprogram must also validate data input and display an error message when the user enters any negative values.*

ii) Write a subprogram named `findAmount` *that accepts consumed kWh through its formal argument list and then returns the total amount to pay (according to the table above).*

iii) Using the subprograms cited above, write a Java program that prompts the user to enter the total number of kWh consumed and then displays the total amount to pay. The program must iterate as many times as the user wishes. At the end of each calculation, the program must ask the user if they wish to calculate the total amount to pay for another consumer. If the answer is "yes" the program must repeat; it must end otherwise. Make your program accept the answer in all possible forms such as "yes", "YES", "Yes", or even "YeS".

Please note that the rates are progressive and that transmission services and distribution charges, as well as federal, state, and local taxes, add a total of 26% to each bill.

Solution

There is nothing new here. Processing progressive rates is something that you have already learned! If this doesn't ring any bells, you need to refresh your memory and review the corresponding Exercise 23.4-5.

The Java program is as follows.

Project_38.2-3

```
static int getConsumption() {
   int consumption ;
   System.out.print("Enter kWh consumed: "); consumption =
   Integer.parseInt(cin.nextLine()); while (consumption < 0) {
      System.out.println("Error: Invalid number!");
      System.out.print("Enter a non-negative number: ");
      consumption = Integer.parseInt(cin.nextLine());
   }
   return consumption; }
static double findAmount(int kwh) {
   double amount;
   if (kwh <= 400) {
      amount = kwh * 0.08;
   }
   else if (kwh <= 1500) {
      amount = 400 * 0.08 + (kwh - 400) * 0.22;
   }
   else if (kwh <= 2000) {
      amount = 400 * 0.08 + 1100 * 0.22 + (kwh - 1500) * 0.35;
   }
   else {
      amount = 400 * 0.08 + 1100 * 0.22 + 500 * 0.35 + (kwh - 2000) * 0.50;
   }
   amount += 0.26 * amount; return amount; }
public static void main(String[] args) throws Exception {
   int kwh; String answer;
   do {
      kwh = getConsumption();
      System.out.println("You need to pay: " + findAmount(kwh));
      System.out.print("Would you like to repeat? ");
      answer = cin.nextLine();
   } while (answer.toUpperCase().equals("YES")); }
```

Exercise 38.2-4 Roll, Roll, Roll the… Dice!

Do the following: i) Write a subprogram named `dice` *that returns a random integer between 1 and 6.*

ii) Write a subprogram named `searchAndCount` *that accepts an integer and an array of integers through its formal argument list and returns the number of times the integer exists in the array.*

iii) Using the subprograms cited above, write a Java program that fills an array with 100 random integers (between 1 and 6) and then lets the user enter an integer. The program must display how many times that user-provided integer exists in the array.

Solution

Both subprograms can be written as methods because they both return one value each. Method `dice()` returns a random integer between 1 and 6, and method `searchAndCount()` returns a number that indicates the number of times an integer exists in an array. The solution is presented here.

Project_38.2-4

```
static final int ELEMENTS = 100;
static int dice() {
   return 1 + (int)(Math.random() * 5); }
static int searchAndCount(int x, int[] a) {
   int count = 0; int i;
   for (i = 0; i <= ELEMENTS - 1; i++) {
      if (a[i] == x) {
         count++;
      }
   }
   return count; }
public static void main(String[] args) throws Exception {
   int x, i;
   int[] a = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      a[i] = dice();
   }
   x = Integer.parseInt(cin.nextLine()); System.out.println("Provided value exists in
   the array " + searchAndCount(x, a) + " times"); }
```

Exercise 38.2-5 How Many Times Does Each Number of the Dice Appear?

Using the methods `dice()` *and* `searchAndCount()` *cited in the previous exercise (Exercise 38.2-4), write a Java program that fills an array with 100 random integers (between 1 and 6) and then displays how many times each of the six numbers appears in the array, as well as which number appears most often.*

Solution

If you were to solve this exercise without using a loop control structure, you would proceed as follows.

```
//Assign to n1 the number of times that value 1 exists in array a n1 =
searchAndCount(1, a); //Assign to n2 the number of times that value 2 exists in array
a n2 = searchAndCount(2, a); .
.
.
//Assign to n6 the number of times that value 6 exists in array a n6 =
searchAndCount(6, a);
//Display how many times each of the six numbers appears in array a
System.out.println(n1 + " " + n2 + " " + n3 + " " + n4 + " " + n5 + " " + n6);
//Find maximum of n1, n2,… n6
maximum = n1;
maxI = 1;
if (n2 > maximum) {
   maximum = n2;
   maxI = 2;
}
if (n3 > maximum) {
   maximum = n3;
   maxI = 3;
}
.
.
.
if (n6 > maximum) {
   maximum = n6;
   maxI = 6;
}
//Display which number appears in the array most often.
System.out.println(maxI);
```

But now that you are reaching the end of this book, of course, you can do something more creative. Instead of assigning each result of the `searchAndCount()` method to individual variables `n1`, `n2`, `n3`, `n4`, `n5`, and

n6, you can assign those results to the positions 0, 1, 2, 3, 4, and 5 of an array named n, as shown here.

```
int[] n = new int[6]; for (i = 0; i <= 5; i++) {
   n[i] = searchAndCount(i + 1, a); }
```

After that, you can find the maximum of the array n using what you learned in Section 34.3.

The complete solution is shown here.

Project_38.2-5

```
static final int ELEMENTS = 100;
static int dice() {
   return 1 + (int)(Math.random() * 5); }
static int searchAndCount(int x, int[] a) {
   int count = 0; int i;
   for (i = 0; i <= ELEMENTS - 1; i++) {
      if (a[i] == x) {
         count++;
      }
   }
   return count; }
public static void main(String[] args) throws Exception {
   int i, maximum, maxI;
   //Create array a of random integers between 1 and 6
   int[] a = new int[ELEMENTS]; for (i = 0; i <= ELEMENTS - 1; i++) {
      a[i] = dice();
   }
   //Create array n and display how many times each of the six numbers appears in
   array a int[] n = new int[6]; for (i = 0; i <= 5; i++) {
      n[i] = searchAndCount(i + 1, a);
      System.out.println("Value " + (i + 1) + " appears " + n[i] + " times");
   }
   //Find maximum of array n maximum = n[0];
   maxI = 0;
   for (i = 1; i <= 5; i++) {
      if (n[i] > maximum) {
         maximum = n[i];
         maxI = i;
      }
   }
   //Display which number appears in the array most often.
   System.out.println("Value " + (maxI + 1) + " appears in the array " + maximum + "
   times."); }
```

38.3 Review Exercises

Complete the following exercises.

1) Do the following: i) Write a subprogram named `displayMenu` that displays the following menu.

   1) Convert USD to Euro (EUR) 2) Convert USD to British Pound Sterling (GBP) 3) Convert USD to Japanese Yen (JPY) 4) Convert USD to Canadian Dollar (CAD) 5) Exit

   ii) Write four different subprograms named `USD_to_EU`, `USD_to_GBP`, `USD_to_JPY`, and `USD_to_CAD`, that accept a currency through their formal argument list and then return the corresponding converted value.

   iii) Using the subprograms cited above, write a Java program that displays the previously mentioned menu and then prompts the user to enter a choice (of 1, 2, 3, 4, or 5). If choice 1, 2, 3, or 4 is selected, the program must prompt the user to enter an amount of money and then it must calculate and display the corresponding converted value. The process must repeat as many times as the user wishes.

   It is given that: ► $1 = 0.94 EUR (€) ► $1 = 0.81 GBP (£) ► $1 = ¥ 149.11 JPY

   ► $1 = 1.36 CAD ($) 2) Do the following: i) Write a subprogram named `displayMenu` that displays the following menu.

   1) Convert USD to Euro (EUR) 2) Convert USD to British Pound Sterling (GBP) 3) Convert EUR to USD

   4) Convert EUR to GBP

   5) Convert GBP to USD

   6) Convert GBP to EUR

   7) Exit

   ii) Write two different subprograms named `USD_to_EUR`, and `USD_to_GBP`, that accept a currency through their formal argument list and then return the corresponding converted value.

   iii) Using the subprograms cited above, write a Java program that displays the previously mentioned menu and then prompts the

user to enter a choice (of 1 to 7) and an amount. The program must then display the required value. The process must repeat as many times as the user wishes. It is given that ► $1 = 0.94 EUR (€) ► $1 = 0.81 GBP (£)

3) Do the following:

i) Write a subprogram named `factorial` that accepts an integer through its formal argument list and returns its factorial.

ii) Using the subprogram `factorial()` cited above, write a subprogram named `mySin` that accepts a value through its formal argument list and returns the sine of *x*, using the Taylor series (shown next) with an accuracy of 0.0000000001.

$$sinx = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

Hint: Keep in mind that *x* is in radians, and $\frac{x^1}{1!} = x$.

iii) Write a subprogram named `degreesToRad` that accepts an angle in degrees through its formal argument list and returns its radian equivalent. It is given that $2\pi = 360^o$.

iv) Using the subprograms `mySin()` and `degreesToRad()` cited above, write a Java program that displays the sine of all integers from 0^o to 360^o.

4) Do the following:

i) Write a subprogram named `isLeap` that accepts a year through its formal argument list and returns `true` or `false` depending on whether or not that year is a leap year.

ii) Write a subprogram named `numOfDays` that accepts a month and a year and returns the number of the days in that month. If that month is February and the year is a leap year, the subprogram must return the value of 29.

Hint: Use the subprogram `isLeap()` cited above.

iii) Write a subprogram named `checkDate` that accepts a day, a month, and a year and returns `true` or `false` depending on whether or not that date is valid.

iv) Using the subprograms cited above, write a Java program that prompts the user to enter a date (a day, a month, and a year) and then calculates and displays the number of days that have passed

between the beginning of the user-provided year and the user-provided date. Using a loop control structure, the program must also validate data input and display an error message when the user enters any non-valid date.

5) In a computer game, players roll two dice. The player who gets the greatest sum of dice gets one point. After ten rolls, the player that wins is the one with the greatest sum of points. Do the following:

i) Write a subprogram named `dice` that returns a random integer between 1 and 6.

ii) Using the subprogram cited above, write a Java program that prompts two players to enter their names. Then, each player consecutively "rolls" two dice ten times. The player that wins is the one with the greatest sum of points.

6) The LAV Car Rental Company has rented 40 cars, which are divided into three categories: hybrid, gas, and diesel. The company charges for a car according to the following table.

Days	Car Type		
	Gas	Diesel	Hybrid
1 - 5	$24 per day	$28 per day	$30 per day
6 - 8	$22 per day	$25 per day	$28 per day
9 and above	$18 per day	$21 per day	$23 per day

Do the following: i) Write a subprogram named `getChoice` that displays the following menu.

1) Gas 2) Diesel 3) Hybrid The subprogram then prompts the user to enter the type of the car (1, 2, or 3) and returns it to the caller.

ii) Write a subprogram named `getDays` that prompts the user to enter the total number of rental days and returns it to the caller.

iii) Write a subprogram named `getCharge` that accepts the type of the car (1, 2, or 3) and the total number of rental days through its formal argument list and

then returns the amount of money to pay according to the previous table. Federal, state, and local taxes add a total of 10% to each bill.

iv) Using the subprograms cited above, write a Java program that prompts the user to enter all necessary information about the rented cars and then displays the following: a) for each car, the total amount to pay including taxes b) the total number of hybrid cars rented c) the total net profit the company gets after removing taxes Please note that the rates are progressive.

7) TAM (Television Audience Measurement) is the specialized branch of media research dedicated to quantify and qualify television audience information.

The LAV Television Audience Measurement Company counts the number of viewers of the main news program on each of 10 different TV channels. The company needs a software application in order to get some useful information. Do the following:

i) Write a subprogram named `getData` that prompts the user to enter into two arrays the names of the channels and the number of viewers of the main news program for each day of the week (Monday to Sunday). It then returns these arrays to the caller.

ii) Write a subprogram `getAverage` that accepts a one-dimensional array through its formal argument list and returns the average value of the first five elements.

iii) Using the subprograms cited above, write a Java program that prompts the user to enter the names of the channels and the number of viewers for each day of the week and then displays the

following:

a) the name of the channels whose average viewer numbers on the weekend were at least 20% higher than the average viewer numbers during the rest of the week.

b) the name of the channels (if any) that, from day to day, showed constantly increasing viewer numbers. If there is no such channel, a corresponding message must be displayed.

8) A public opinion polling company asks 300 citizens whether they have been hospitalized during the Covid-19 lockdown period. Do the following:

i) Write a subprogram named `inputData` that prompts the user to enter the citizen's SSN (Social Security Number) and their answer (Yes, No) into two arrays, `SSNs` and `answers`, respectively. The two arrays must be returned to the caller.

ii) Write a subprogram named `sortArrays` that accepts the arrays `SSNs` and `answers` through its formal argument list. It then sorts array `SSNs` in ascending order using the selection sort algorithm. The subprogram must preserve the one-to-one correspondence between the elements of the two arrays.

iii) Write a subprogram named `searchArray` that accepts array `SSNs` and an SSN through its formal argument list and then returns the index position of that SSN in the array. If the SSN is not found, a message “SSN not found” must be displayed and the value −1 must be returned. Use the binary search algorithm.

iv) Write a subprogram named `countAnswers` that accepts the array `answers` and an answer through its formal argument list. It then returns the number of times this answer exists in the array.

v) Using the subprograms cited above, write a Java program that prompts the user to enter the SSNs

and the answers of the citizens. It must then prompt the user to enter an SSN and display the answer that the citizen with this SSN gave, as well as the percentage of citizens that gave the same answer in relation to the total number of citizens. The program must then ask the user if they wish to search for another SSN. If the answer is “Yes” the process must repeat; it must end otherwise.

9) Eight teams participate in a football tournament, and each team plays 12 games, one game each week. Do the following:

i) Write a subprogram named `inputData` that prompts the user to enter the name of each team and the letter “W” for win, “L” for loss, or “T” for tie (draw) for each game into two arrays, `names` and `results`, respectively. It then returns the arrays to the caller.

ii) Write a subprogram named `displayResult` that accepts arrays `names` and `results` through its formal argument list. It then prompts the user for a letter (W, L, or T) and displays, for each team, the week number(s) in which the team won, lost, or tied respectively. For example, if the user enters “L”, the subprogram must search and display, for each team, the week numbers (e.g., week 3, week 14, and so on) in which the team lost the game.

iii) Write a subprogram named `findTeam` that accepts array `names` through its formal argument list. It then prompts the user to enter the name of a team and returns the index position of that team in the array. If the user-provided team name does not exist, the value −1 must be returned.

iv) Using the subprograms cited above, write a Java program that prompts the user to enter the name of each team and the letter “W” for win, “L” for loss, or “T” for tie (draw) for each game. It must then prompt the user for a letter (W, L, or T) and

display, for each team, the week number(s) in which the team won, lost, or tied respectively.

Finally, the program must prompt the user to enter the name of a team. If the user-provided team is found, the program must display the total number of points for this team and then prompt the user to enter the name of another team. This process must repeat as long as the user enters an existing team name. If user-provided team name is not found, the message “Team not found” must be displayed and the program must end.

It is given that a win receives 3 points and a tie receives 1 point.

10) Do the following:

i) Write a subprogram named `hasDuplicateDigits` that accepts an integer and returns `true` when any of its digits appears more than once; it must return `false` otherwise.

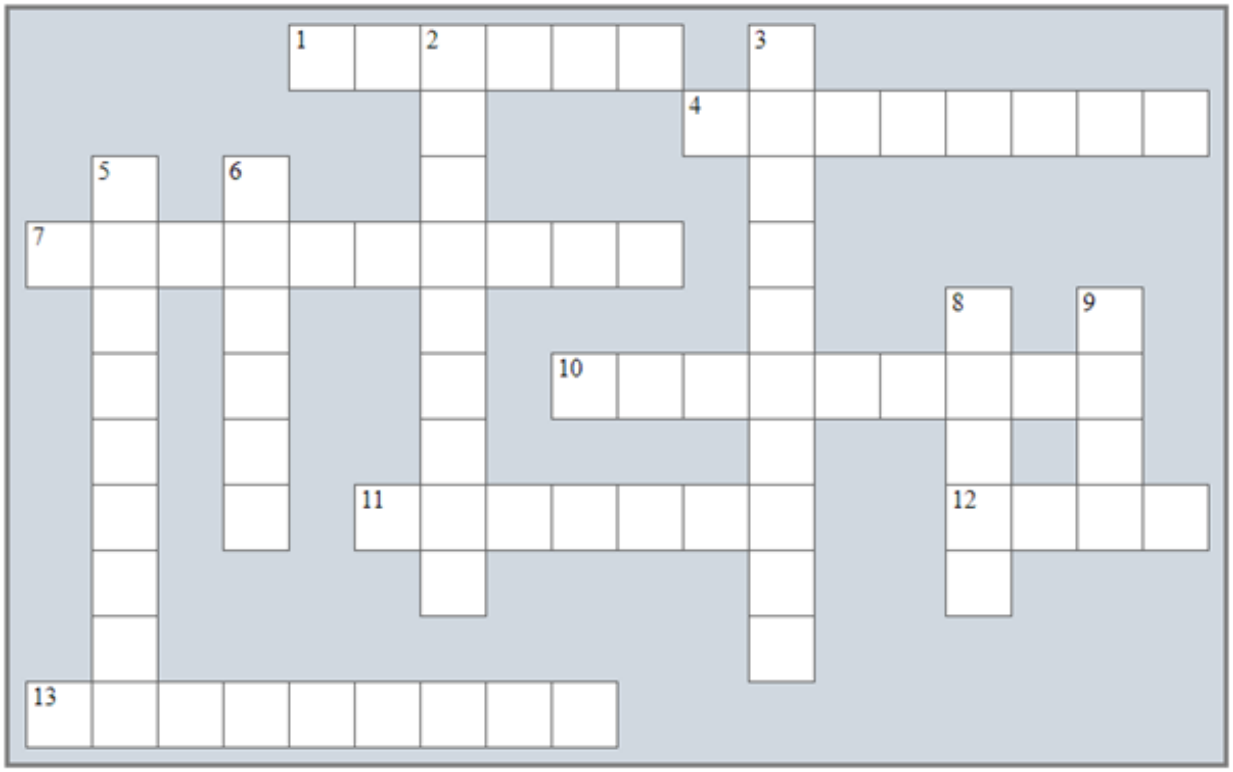
Hint: Declare an array of 10 elements to keep track of the occurrences of each digit. The array must be initialized to all zeros.

ii) Using the subprogram cited above, write a Java program that prompts the user to enter an integer and displays a message indicating whether or not, any of its digits appears more than once. Moreover, using a loop control structure, the program must validate data input and display an error message when the user enters any value less than 11.

Review in “Subprograms”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

1) A method may contain an argument list called a _____ argument list.

4) Generally speaking, this subprogram returns a result.

7) In this kind of programming, a problem is subdivided into smaller subproblems.

10) A sequence of numbers where the first two numbers are 0 and 1, and each subsequent number is the sum of the previous two.

11) In this kind of programming, subprograms of common functionality are grouped together into separate modules.

12) Send a value to a method.

13) Arrays in Java are passed by _____.

Down

2) A programming technique in which a subprogram calls itself.

3) A block of statements packaged as a unit that performs a specific task.

5) Generally speaking, this subprogram returns no result.

6) When a subprogram is called, the passed argument list is called an _____ argument list.

8) It refers to the range of effect of a variable.

9) The principle which states that most systems work best if they are kept simple, avoiding any unnecessary complexity!

Review Questions

Answer the following questions.

1) What is a subprogram? Name some built-in subprograms of Java.
2) What is procedural programming?
3) What are the advantages of procedural programming?
4) What is modular programming? Name one module of Java you know.
5) What is the general form of a Java method?
6) How do you make a call to a method?
7) Describe the steps that are performed when the main code makes a call to a method.
8) What is a void method?
9) What is the general form of a Java void method?
10) How do you make a call to a void method?
11) Describe the steps that are performed when the main code makes a call to a void method.
12) What is the difference between a method and a void method?
13) What is the formal argument list?
14) What is the actual argument list?
15) Can two subprograms use variables of the same name?
16) How long does a subprogram's variable “live” in main memory?
17) How long does a main code's variable “live” in main memory?

18) Can a subprogram call another subprogram? If yes, give some examples.
19) What does it mean to “pass an argument by value”?
20) What does it mean to “pass an argument by reference”?
21) What is an optional argument?
22) What is meant by the term “scope” of a variable?
23) What happens when a variable has a local scope?
24) What happens when a variable has a global scope?
25) What is the difference between a local and a global variable?
26) What is recursion?
27) What are the three rules that all recursive algorithms must follow?

Part VIII
Object-Oriented Programming

Chapter 39
Introduction to Object-Oriented Programming

39.1 What is Object-Oriented Programming?

In Part VII all the programs that you read or even wrote, were using subprograms (methods and void methods). This programming style is called procedural programming and most of the time it is just fine! But when it comes to writing large programs, or working in a big company such as Microsoft, Facebook, or Google, object-oriented programming is a must use programming style!

Object-oriented programming, usually referred to as OOP, is a style of programming that focuses on *objects*. In OOP, data and functionality are combined and encapsulated inside something called an object. Applying object-oriented programming principles enables you to maintain your code more easily, and write code that can be easily understood and used by others.

What does the statement "*OOP focuses on objects*" truly mean? Let's consider an example from the real world. Imagine a car. How would you describe a particular car? It has specific attributes, such as the brand, the model, the color, and the license plate. Additionally, there are specific actions this car can perform, or have performed on it. For instance, someone can turn it on or off, accelerate or apply the brakes, or park.

In OOP, this car can be represented as an object with specific attributes (commonly referred to as *fields*) that can perform specific actions (referred to as *methods*).

Obviously, you may now be asking yourself, "*How can I create objects in the first place?*" The answer is simple! All you need is a *class*. A class resembles a "rubber inkpad stamp"! In **Figure 39-1** there is a stamp (this is the class) with four empty fields and three actions (methods).

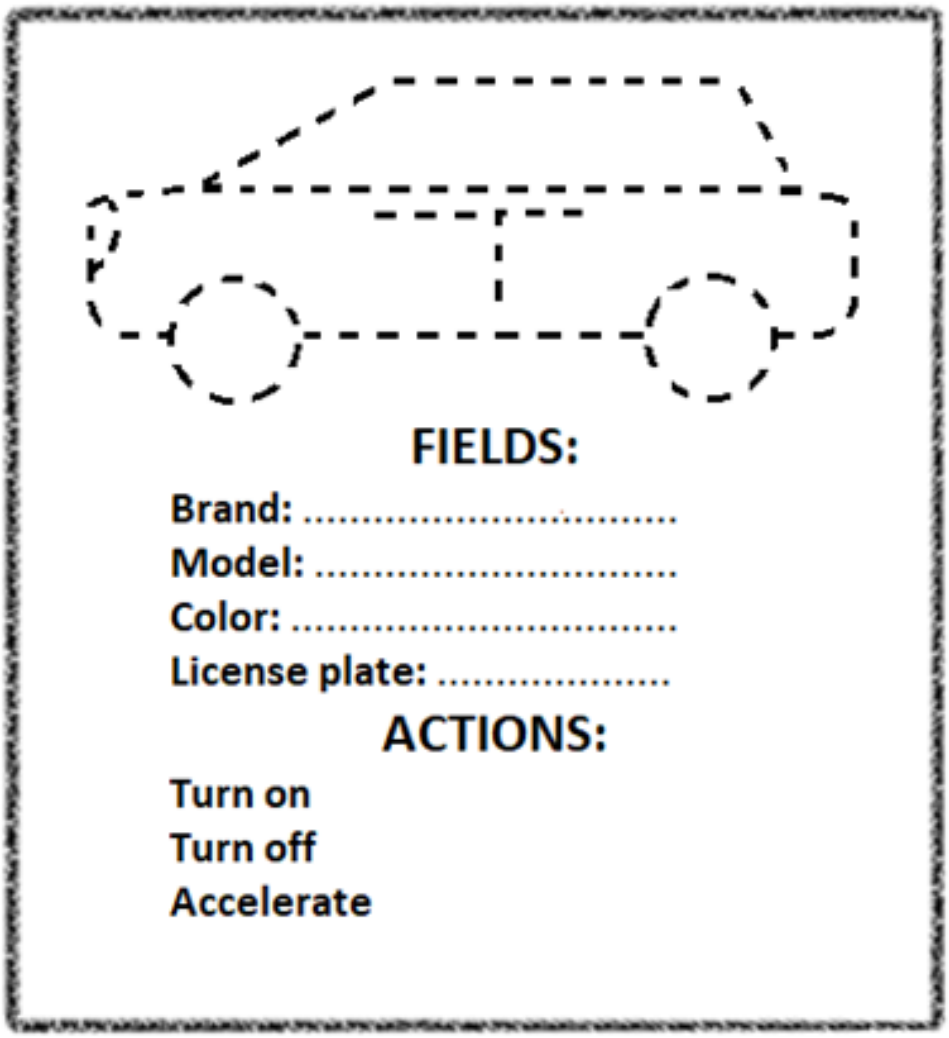


Figure 39-1 A class resembles a “rubber inkpad stamp”

Someone who uses this stamp can stamp-out many cars (these are the objects). In **Figure 39-2**, for example, a little boy stamped-out those two cars and then he colored them and filled out each car's fields with specific attributes.

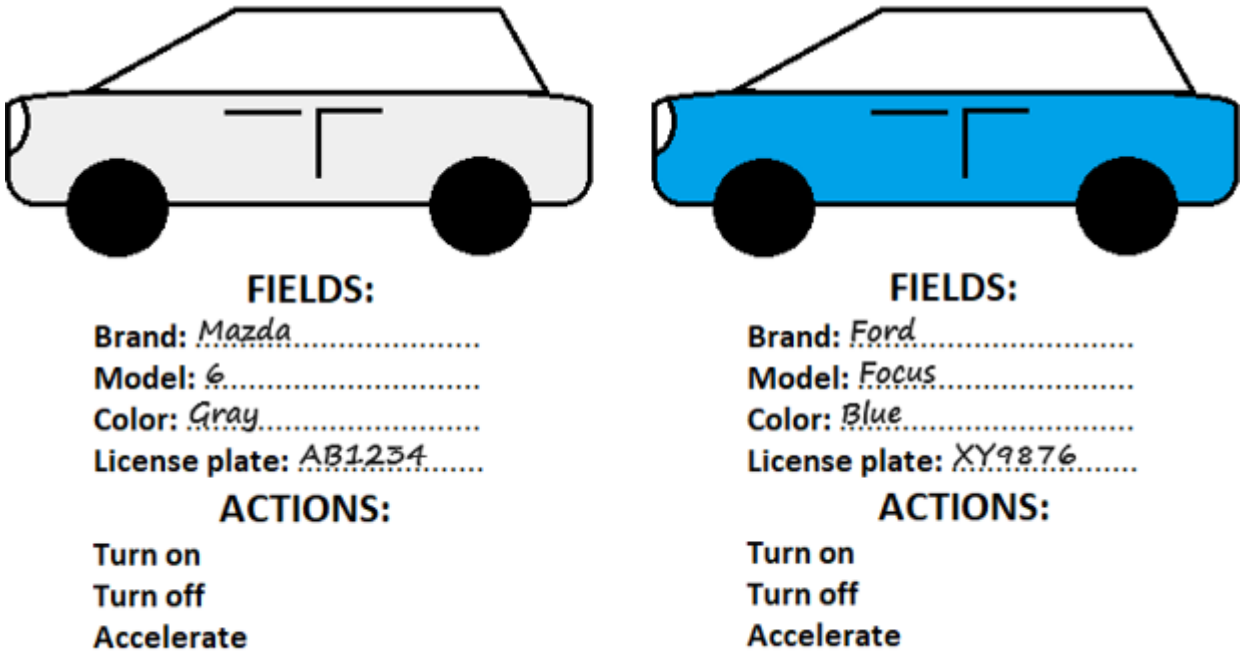


Figure 39-2 You can use the same rubber stamp as a template to stamp-out many cars

✎ The process of creating a new object (a new instance of a class) is called "instantiation".

✎ A class is a template and every object is created from a class. Each class should be designed to carry out one, and only one, task! This is why, most of the time, more than one class is used to build an entire application!

✎ In OOP, the rubber stamp is the class. You can use the same class as a template to create (instantiate) many objects!

39.2 Classes and Objects in Java

Now that you have a grasp of the theoretical concepts behind classes and objects, let's dive into writing a real class in Java! The following code fragment creates the class `Car`. There are four fields and three methods within the class.

```
class Car {
   //Define four fields (attributes) public String brand = ""; public String model =
   ""; public String color = ""; public String licensePlate = "";
   //Define method turnOn() public void turnOn() {
      System.out.println("The car turns on");
   }
   //Define method turnOff() public void turnOff() {
      System.out.println("The car turns off");
   }
   //Define method accelerate() public void accelerate() {
```

```
        System.out.println("The car accelerates");
    }
}
```

And here's an interesting tidbit: Fields and methods within classes are essentially just ordinary variables and subprograms respectively!

✎ *The class* `Car` *is just a template. No objects are created yet!*

✎ *The keyword* `public` *in front of a field or method specifies that this field or method can be accessed from outside the class using an instance of the class.*

✎ *The name of a class should follow the Upper Camel Case convention as well as all the rules for naming variables presented in* *Section 5.4**.*

To create two objects (or in other words to create two instances of the class `Car`), you need the following two lines of code.

```
Car car1 = new Car(); Car car2 = new Car();
```

✎ *An object is nothing more than an instance of a class, and this is why, many times, it may be called a "class instance" or "class object".*

When you create a new object (a new instance of a class) the process is called "instantiation".

Now that you have created (instantiated) two objects, you can assign values to their fields. To do so, use the dot notation. This means you need to write the name of the object, followed by a dot and then the name of the field or method you want to access. The following code fragment creates two objects, `car1` and `car2`, and assigns values to their fields.

```
public static void main(String[] args) throws Exception {
    Car car1 = new Car(); Car car2 = new Car();
    car1.brand = "Mazda"; car1.model = "6"; car1.color = "Gray"; car1.licensePlate =
    "AB1234";
    car2.brand = "Ford"; car2.model = "Focus"; car2.color = "Blue"; car2.licensePlate =
    "XY9876";
    System.out.println(car1.brand); //It displays: Mazda
    System.out.println(car2.brand); //It displays: Ford }
```

✎ *In the previous example,* `car1` *and* `car2` *are two instances of the same class. Using* `car1` *and* `car2` *with dot notation allows you to refer to only one*

instance at a time. If you make any changes to one instance they will not affect the other instance!

The next code fragment calls the methods `turnOff()` and `accelerate()` of the objects `car1` and `car2` respectively.

```
car1.turnOff(); car2.accelerate();
```

A class is a template that cannot be executed, whereas an object is an instance of a class that can be executed!

One class can be used to create (instantiate) as many objects as you want!

39.3 The Constructor and the Keyword `this`

In Java, there is a method that has a special role and is called *constructor*. The constructor method is executed automatically whenever an instance of a class (an object) is created. Any initialization that you want to do with your object can be done within this method. In Java, the constructor is a method whose name is the same as the name of its class.

Take a look at the following example. The constructor method `Person()` is called twice automatically, once when the object `p1` is created and once when the object `p2` is created, which means that the message "An object was created" is displayed twice.

```
Project_39.3a
class Person {
//Define the constructor public Person() {
System.out.println("An object was created");
}
}
public class App {
public static void main(String[] args) throws
Exception {
Person p1 = new Person();
Person p2 = new Person();
}
}
```

✎ *Note that there is no keyword* `void` *in front of the name of the constructor.*

In object-oriented programming (OOP) with Java, there is a keyword named `this`, which serves as a reference variable pointing to the current object. Take a look at the following example.

```
🗀 Project_39.3b
class Person {
public String name; public int age;
//Define the constructor public Person() {
System.out.println("An object was created");
}
public void sayInfo() {
System.out.println("I am " + this.name);
System.out.println("I am " + this.age + " years
old");
}
}
public class App {
public static void main(String[] args) throws
Exception {
Person person1 = new Person();
//Assign values to its fields
person1.name = "John";
person1.age = 14;
person1.sayInfo(); //Call the method sayInfo() of
the object person1
}
}
```

✎ *Note that when declaring the fields* `name` *and* `age` *outside of a method* (*but within the class*), *you need to write the field name without dot notation. To access the fields, however, from within a method, you should use dot notation* (*for example,* `this.name` *and* `this.age`) .

A question that is probably spinning around in your head right now is "*Why is it necessary to refer to these fields* `name` *and* `age` *within the method* `sayInfo()` *as* `this.name` *and* `this.age`*? Is it really necessary to use the*

keyword `this` *in front of them?*" A simple answer is that there is always a possibility that you could have two extra local variables of the same name (`name` and `age`) within the method. So you need a way to distinguish among those local variables and the object's fields. If you are confused, try to understand the following example. There is a field `b` within the class `MyClass` and a local variable `b` within the method `myMethod()` of the class. The `this` keyword is used to differentiate between the local variable and the field.

Project_39.3c

```
class FooClass {
public String b; //This is a field
public void myMethod() {
String b = " *** "; //This is a local variable
System.out.println(b + this.b + b);
}
}
public class App {
public static void main(String[] args) throws
Exception {
MyClass x = new FooClass(); //Create object x
x.b = "Hello!"; //Assign a value to its field
x.myMethod(); //It displays: *** Hello! ***
}
}
```

✎ *The keyword* `this` *can be used to refer to any member* (*field or method*) *of a class from within a method of the class.*

39.4 Passing Initial Values to the Constructor

Any method, even the constructor, method can have formal arguments within its formal argument list. For example, in the constructor method you can use arguments to pass some initial values to the object during creation. The example that follows creates four objects, each of which represents a Titan[24] from Greek mythology.

Project_39.4

```
class Titan {
```

```
   public String name; public String gender;
   //Define the constructor public Titan(String n, String g) {
      this.name = n;
      this.gender = g;
   }
}
public class App {
   public static void main(String[] args) throws Exception {
      Titan titan1 = new Titan("Cronus", "male");
      Titan titan2 = new Titan("Oceanus", "male");
      Titan titan3 = new Titan("Rhea", "female");
      Titan titan4 = new Titan("Phoebe", "female");
   }
}
```

In Java, it is legal to have one field and one local variable (or even a formal argument) with the same name. So, the class `Titan` can also be written as follows

```
class Titan {
    public String name; public String gender;
    //Define the constructor public Titan(String name, String gender) {
        this.name = name; //Fields and arguments can have the same name
        this.gender = gender;
    }
}
```

The variables `name` and `gender` are arguments used to pass values to the constructor whereas `this.name` and `this.gender` are fields used to store values within the object.

Exercise 39.4-1 Historical Events

Do the following: i) Write a class named `HistoryEvents` *which includes a) a public string field named* `day`*.*

b) a public string array field named `events` *of size 2.*

c) a constructor that accepts an initial value for the field `day` *through its formal argument list.*

ii) Write a Java program that creates two objects of the class `HistoryEvents` *for the following historical events: 4th of July 1776: Declaration of Independence in United States; 28th of October 969: Byzantine troops occupy Antioch; 1810: French troops occupy Amsterdam; 1940: Ohi Day in Greece and then displays all available information.*

Solution

The solution is as follows.

Project_39.4-1

```
class HistoryEvents {
   public String day; public String[] events = new String[2];
   //Define the constructor public HistoryEvents (String day) {
         this.day = day;
   }
}
public class App {
   public static void main(String[] args) throws Exception {
      HistoryEvents h1 = new HistoryEvents("4th of July");
      h1.events[0] = "1776: Declaration of Independence in United States";
      h1.events[1] = "1810: French troops occupy Amsterdam";
      HistoryEvents h2 = new HistoryEvents("28th of October");
      h2.events[0] = "969: Byzantine troops occupy Antioch";
      h2.events[1] = "1940: Ohi Day in Greece";
      System.out.println(h1.day);
      System.out.println(h1.events[0]);
      System.out.println(h1.events[1]);
      System.out.println();
      System.out.println(h2.day);
      System.out.println(h2.events[0]);
      System.out.println(h2.events[1]);
   }
}
```

39.5 Getter and Setter Methods

A field is a variable declared directly in a class. The principles of the object-oriented programming, though, state that the data of a class should be hidden and safe from accidental alteration. Think that one day you will probably be writing classes that other programmers will use in their programs. So, you don't want them to know what is inside your classes! The internal operation of your classes should be kept hidden from the outside world. By not exposing a field, you manage to hide the internal implementation of your class. Fields should be kept private to a class and accessed through get and set methods.

✎ *Generally speaking, programmers should use fields only for data that have private or protected accessibility. In Java you can set a*

field (or a method) as private or protected using the special keywords `private` *or* `protected` *correspondingly.*

Let's try to understand all of this new stuff through an example. Suppose you write the following class that converts a degrees Fahrenheit temperature into its degrees Celsius equivalent.

```
Project_39.5a
class FahrenheitToCelsius {
  public double temperature;
//Define the constructor public
FahrenheitToCelsius(double value) {
 this.temperature = value; //Field is
                initialized
                }
//This method gets the temperature public double
                getTemperature() {
   return 5.0 / 9.0 * (this.temperature - 32.0);
                }
               }
            public class App {
 public static void main(String[] args) throws
                Exception {
            FahrenheitToCelsius x = new
  FahrenheitToCelsius(-68); //Create object x
    System.out.println(x.getTemperature());
                }
               }
```

This class is almost perfect but has a main disadvantage. It doesn't take into consideration that a temperature cannot go below −459.67 degrees Fahrenheit (−273.15 degrees Celsius). This temperature is called absolute zero. So a novice programmer who uses your class and knows absolutely nothing about physics, might pass a value of −500 degrees Fahrenheit to the constructor, as shown in the code fragment that follows

FahrenheitToCelsius *x = new* ***FahrenheitToCelsius****(-500);*
*System.out.****println****(x.****getTemperature****());*

Even though the program can run perfectly well and display a value of −295.55 degrees Celsius, unfortunately this temperature cannot exist in the entire universe! So a slightly different version of this class might partially solve the problem.

```
Project_39.5b
class FahrenheitToCelsius {
  public double temperature;
//Define the constructor public
FahrenheitToCelsius(double value) throws Exception {
   this.setTemperature(value); //Use a method to set
        the value of the field temperature
                        }
  //This method gets the temperature public double
                  getTemperature() {
    return 5.0 / 9.0 * (this.temperature - 32.0);
                        }
   //This method sets the temperature public void
   setTemperature(double value) throws Exception {
                if (value >= -459.67) {
                 this.temperature = value;
                          }
                        else {
      throw new Exception("There is no temperature
                   below -459.67");
                          }
                        }
                      }
                public class App {
   public static void main(String[] args) throws
                     Exception {
              FahrenheitToCelsius x = new
     FahrenheitToCelsius(-500); //Create object x
       System.out.println(x.getTemperature());
                        }
                      }
```

✎ *The* `throw` *statement forces the program to throw an exception (a runtime error) causing the flow of execution to stop.*

This time, a method named `setTemperature()` is used to set the value of the field `temperature`. This is better, but not exactly perfect, because the programmer must be careful and always remember to use this method each time they wish to change the value of the field `temperature`. The problem is that the value of the field `temperature` can still be directly changed using its name, as shown in the code fragment that follows.

```
FahrenheitToCelsius x = new FahrenheitToCelsius(-50); //Create object x
System.out.println(x.getTemperature());
x.setTemperature(-65); //This is okay!
System.out.println(x.getTemperature());
x.temperature = -500; //Unfortunately, this is still permitted!
System.out.println(x.getTemperature());
```

This problem can be completely solved if you declare the field `temperature` as private! When a field is declared as private, the caller (here the object `x`) cannot get direct access to the field, as shown in the Java program that follows.

🗀 **Project_39.5c**

```
class FahrenheitToCelsius {
    private double _temperature; //Declare field temperature as private
    //Define the constructor public FahrenheitToCelsius(double value) throws Exception {
        this.setTemperature(value); //Call the setter
    }
    //This method gets the temperature public double getTemperature() {
        return 5.0 / 9.0 * (this._temperature - 32.0);
    }
    //This method sets the temperature public void setTemperature(double value) throws Exception {
        if (value >= -459.67) {
            this._temperature = value;
        }
        else {
```

```
            throw new Exception("There is no temperature
                        below -459.67");
                    }
                }
            }
        public class App {
    public static void main(String[] args) throws
                    Exception {
            FahrenheitToCelsius x = new
    FahrenheitToCelsius(-50); //Create object x. This
                        calls the
                        //constructor which,
                    //in turn, calls the setter.
     System.out.println(x.getTemperature()); //This
                    calls the getter.
     x.setTemperature(-65); //This calls the setter.
     System.out.println(x.getTemperature()); //This
                    calls the getter.
     x._temperature = -50; //This is NOT permitted!
    System.out.println(x._temperature); //This is NOT
                    permitted as well!
                    }
                }
```

✎ *Many programmers prefer to have private fields prefixed with an underscore (_).*

Exercise 39.5-1 The Roman Numerals

Roman numerals are shown in the following table.

Number	Roman Numeral
1	I
2	II
3	III
4	IV

5	V

Do the following: ii) Write a class named `Romans` *which includes a) a private integer field named* `_number`.

b) a getter and a setter method named `getNumber` *and* `setNumber` *correspondingly. They will be used to get and set the value of the field* `_number` *in integer format. The setter method must throw an error when the number is not recognized.*

c) a getter and a setter method named `getRoman` *and* `setRoman` *correspondingly. They will be used to get and set the value of the field* `_number` *in Roman numeral format. The setter method must throw an error when the Roman numeral is not recognized.*

iii) Using the class cited above, write a Java program that displays the Roman numeral that corresponds to the value of 3 as well as the number that corresponds to the Roman numeral value of "V".

Solution

The getter and setter methods of the field `_number` in integer format are very simple so there is nothing special to explain. The getter and setter methods of the field `_number` in Roman numeral format, however, need some explanation.

The getter method of the field `_number` in Roman numeral format can be written as follows

```
//Define the getter public String getRoman() {
    String retValue = "";
    if (this._number == 1)
    retValue = "I";
    else if (this._number == 2)
    retValue = "II";
    else if (this._number == 3)
    retValue = "III";
    else if (this._number == 4)
    retValue = "IV";
    else if (this._number == 5)
    retValue = "V";
    return retValue; }
```

However, since you now know many about hashmaps, you can use a better approach, as shown in the code fragment that follows.

```
//Define the getter public String getRoman() {
   HashMap<Integer, String> number2roman = new HashMap<>(
      Map.of(1, "I", 2, "II", 3, "III", 4, "IV", 5, "V")
   );
   return number2roman.get(this._number); }
```

Accordingly, the setter method can be as follows

```
//Define the setter public void setRoman(String key) throws Exception {
    HashMap<String, Integer> roman2number = new HashMap<>(
        Map.of("I", 1, "II", 2, "III", 3, "IV", 4, "V", 5)
    );
    if (roman2number.containsKey(key)) {
        this._number = roman2number.get(key);
    }
    else {
        throw new Exception("Roman numeral not recognized");
    }
}
```

The Java built-in method `struct.containsKey(key_name)` *returns* `true` *when the hashmap* `struct` *contains the specified key* `key_name` *within its keys collection.*

The statement `if (roman2number.containsKey(key))` *is equivalent to the statement* `if (roman2number .containsKey(key) == true)`.

The final Java program is as follows Project_39.5-1

```
class Romans {
   private int _number;
   //Define the getter public int getNumber() {
      return this._number;
   }
   //Define the setter public void setNumber(int value) throws Exception {
      if (value >= 1 && value <= 5) {
         this._number = value;
      }
      else {
         throw new Exception("Number not recognized");
      }
   }
```

```
   //Define the getter public String getRoman() {
      HashMap<Integer, String> number2roman = new HashMap<>(
         Map.of(1, "I", 2, "II", 3, "III", 4, "IV", 5, "V")
      );
      return number2roman.get(this._number);
   }
   //Define the setter public void setRoman(String key) throws Exception {
      HashMap<String, Integer> roman2number = new HashMap<>(
         Map.of("I", 1, "II", 2, "III", 3, "IV", 4, "V", 5)
      );
      if (roman2number.containsKey(key)) {
         this._number = roman2number.get(key);
      }
      else {
         throw new Exception("Roman numeral not recognized");
      }
   }
}
public class App {
   public static void main(String[] args) throws Exception {
      Romans x = new Romans();
      x.setNumber(3);
      System.out.println(x.getNumber()); //It displays: 3
      System.out.println(x.getRoman()); //It displays: III
      x.setRoman("V");
      System.out.println(x.getNumber()); //It displays: 5
      System.out.println(x.getRoman()); //It displays: V
   }
}
```

39.6 Can a Method Call Another Method of the Same Class?

In Section 37.2 you learned that a subprogram can call another subprogram. Obviously, the same applies when it comes to class methods—a method can call another method of the same class! Methods are nothing more than subprograms after all! So, if you want a method to call another method of the same class you should use the keyword `this` in front of the method that you want to call (using dot notation) as shown in the example that follows.

Project_39.6

```
class JustAClass {
   public void foo1() {
      System.out.println("foo1 was called");
      this.foo2(); //Call foo2() using dot notation
```

```
    }
    public void foo2() {
        System.out.println("foo2 was called");
    }
}
public class App {
    public static void main(String[] args) throws Exception {
        JustAClass x = new JustAClass();
        x.foo1(); //Call foo1() which, in turn, will call foo2()
    }
}
```

Exercise 39.6-1 Doing Math

Do the following: i) Write a class named `DoingMath` *which includes a) a private void method named* `square` *that accepts a number through its formal argument list and then calculates its square and displays the message "The square of XX is YY", where XX and YY must be replaced by actual values.*

b) a private void method named `squareRoot` *that accepts a number through its formal argument list and then calculates its square root and displays the message "The square root of XX is YY" where XX and YY must be replaced by actual values. However, if the number is less than zero, the method must display an error message.*

c) a public void method named `displayResults` *that accepts a number through its formal argument list and then calls the methods* `square()` *and* `squareRoot()` *to display the results.*

ii) Using the class cited above, write a Java program that prompts the user to enter a number. The program must then display the root and the square root of that number.

Solution

This exercise is quite simple. The methods `square()`, `squareRoot()`, and `displayResults()` must have a formal argument within their formal argument list so as to accept a passed value. The solution is as follows.

Project_39.6-1

```
class DoingMath {
    private void square(double x) { //Argument x accepts passed value
    System.out.println("The square of " + x + " is " + (x * x));
    }
```

```
   private void squareRoot(double x) { //Argument x accepts passed value
   if (x < 0) {
          System.out.println("Cannot calculate square root");
      }
      else {
          System.out.println("Square root of " + x + " is " + Math.sqrt(x));
      }
   }
   public void displayResults(double x) { //Argument x accepts passed value
   this.square(x);
      this.squareRoot(x);
   }
}
public class App {
   static Scanner cin = new Scanner(System.in);
   public static void main(String[] args) throws Exception {
      double b;
      DoingMath dm = new DoingMath();
      System.out.print("Enter a number: ");
      b = Double.parseDouble(cin.nextLine());
      dm.displayResults(b);
   }
}
```

39.7 Class Inheritance

Class inheritance is one of the main concepts of OOP. It lets you write a class using another class as a base. When a class is based on another class, the programmers use to say “it inherits the other class”. The class that is inherited is called the *parent class*, the *base class*, or the *superclass*. The class that does the inheriting is called the *child class*, the *derived class*, or the *subclass*.

A child class automatically inherits all the methods and fields of the parent class. The best part, however, is that you can add additional characteristics (methods or fields) to the child class. Therefore, you use inheritance when you have to write several classes that share many common characteristics but aren't entirely identical. To do this, you work as follows. First, you write a parent class containing all the common characteristics. Next, you write child classes that inherit all those common characteristics from the parent class. Finally, you add any additional and unique characteristics, specific to each child class. Just as with humans, it's these additional and unique characteristics that set a child apart from its parent, right?

Let's say that you want to write a program that keeps track of the teachers and students in a school. They have some characteristics in common, such as name and age, but they also have specific characteristics such as salary for teachers and final grade for students that are not in common. What you can do here is write a parent class named `SchoolMember` that contains all those characteristics that both teachers and students have in common. Then you can write two child classes named `Teacher` and `Student`, one for teachers and one for students. Both child classes can inherit the class `SchoolMember` but additional fields, named `salary` and `finalGrade`, must be added to the child classes `Teacher` and `Student` correspondingly.

The parent class `SchoolMember` is shown here

```
class SchoolMember {
    public String name; public int age;
    //Define the constructor public SchoolMember(String name, int age) {
        this.name = name;
        this.age = age;
        System.out.println("A school member was initialized");
    }
}
```

If you want a class to inherit the class `SchoolMember`, it must be defined as follows

```
class Name extends SchoolMember {
    Define additional fields for this class
    //Define the constructor public Name(String name, int age [, …]) {
        super(name, age); //Call the constructor of the class SchoolMember
        Additional statement or block of statements
    }
    Define additional methods and/or properties for this class
}
```

where *`Name`* is the name of the child class.

So, the class `Teacher` can be as follows

```
class Teacher extends SchoolMember {
    public double salary; //This is an additional field for this class
    //Define the constructor public Teacher(String name, int age, double
    salary) {
        super(name, age); //Call the constructor of the class
        SchoolMember
        this.salary = salary;
        System.out.println("A teacher was initialized");
    }
    //This is an additional method for this class public void displayValues()
    {
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
        System.out.println("Salary: " + this.salary);
    }
}
```

✎ *The statement* `super(name, age)` *calls the constructor of the class* `SchoolMember` *and initializes the fields* `name` *and* `age` *of the class* `Teacher`.

Similarly, the class `Student` can be as follows

```
class Student extends SchoolMember {
    public String finalGrade; //This is an additional field for this class
    //Define the constructor public Student(String name, int age, String
    finalGrade) {
        super(name, age); //Call the constructor of the class
        SchoolMember
        this.finalGrade = finalGrade;
        System.out.println("A student was initialized");
    }
    //This is an additional method for this class public void displayValues()
    {
        System.out.println("Name: " + this.name);
        System.out.println("Age: " + this.age);
        System.out.println("Final grade: " + this.finalGrade);
    }
}
```

✎ *The statement* `super(name, age)` *calls the constructor of the class* `SchoolMember` *and initializes the fields* `name` *and* `age` *of the class* `Student`.

The complete Java program is as follows.

Project_39.7

```
//Define the class SchoolMember.
class SchoolMember {
   public String name; public int age;
   //Define the constructor public SchoolMember(String name, int age) {
      this.name = name;
      this.age = age;
      System.out.println("A school member was initialized");
   }
}
//Define the class Teacher. It inherits the class SchoolMember.
class Teacher extends SchoolMember {
   public double salary; //This is an additional field for class Teacher
   //Define the constructor public Teacher(String name, int age, double salary) {
      super(name, age); //Call the constructor of the class SchoolMember
      this.salary = salary;
      System.out.println("A teacher was initialized");
   }
   //This is an additional method for this class public void displayValues() {
      System.out.println("Name: " + this.name);
      System.out.println("Age: " + this.age);
      System.out.println("Salary: " + this.salary);
   }
}
//Define the class Student. It inherits the class SchoolMember.
class Student extends SchoolMember {
   public String finalGrade; //This is an additional field for this class
   //Define the constructor public Student(String name, int age, String finalGrade) {
      super(name, age); //Call the constructor of the class SchoolMember
      this.finalGrade = finalGrade;
      System.out.println("A student was initialized");
   }
   //This is an additional method for this class public void displayValues() {
      System.out.println("Name: " + this.name);
      System.out.println("Age: " + this.age);
      System.out.println("Salary: " + this.finalGrade);
   }
}
public class App {
```

```
   public static void main(String[] args) throws Exception {
      Teacher teacher1 = new Teacher("Mr. John Scott", 43, 35000);
      Teacher teacher2 = new Teacher("Mrs. Ann Carter", 5, 32000);
      Student student1 = new Student("Mark Nelson", 14, "A");
      Student student2 = new Student("Mary Morgan", 13, "B");
      teacher1.displayValues();
      teacher2.displayValues();
      student1.displayValues();
      student2.displayValues();
   }
}
```

39.8 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) Procedural programming is better than object-oriented programming when it comes to writing large programs.
2) Object-oriented programming focuses on objects.
3) An object combines data and functionality.
4) Object-oriented programming enables you to maintain your code more easily but your code cannot be used easily by others.
5) You can create an object without using a class.
6) The process of creating a new instance of a class is called “installation”.
7) In OOP, you always have to create at least two instances of the same class.
8) The constructor method is executed when an object is instantiated.
9) When you create two instances of the same class, the constructor method of the class will be executed twice.
10) The keyword `private` in front of a field specifies that this field can be accessed from outside the class.
11) The keyword `public` in front of a method specifies that this method can be called from outside the class.
12) The principles of the object-oriented programming state that the data of a class should be hidden and safe from accidental alteration.
13) Getter and setter methods provide a flexible mechanism to read, write, or compute the value of a field.

14) Getter and setter methods expose the internal implementation of a class.
15) Class inheritance is one of the main concepts of OOP.
16) When a class is inherited, it is called the “derived class”.
17) A parent class automatically inherits all the methods and fields of the child class.

39.9 Review Exercises

Complete the following exercises.

1) Do the following:
    i) Write a class named `Geometry` that includes:
        a) a public method named `rectangleArea` that accepts the base and the height of a rectangle through its formal argument list and then calculates and returns its area.
        b) a public method named `triangleArea` that accepts the base and the height of a triangle through its formal argument list and then calculates and returns its area. It is given that

$$Area = \frac{1}{2} Base \times Height$$

    ii) Using the class cited above, write a Java program that prompts the user to enter the side of a square, the base and the height of a rectangle, and the base and the height of a triangle, and then displays the area for each one of them.

2) Do the following:
    i) Write a class named `Pet` which includes:
        a) a public string field named `kind`
        b) a public integer field named `legsNumber`
        c) a public void method named `startRunning` that displays the message “Pet is running”
        d) a public void method named `stopRunning` that displays the message “Pet stopped”
    ii) Write a Java program that creates two instances of the class `Pet` (for example, a dog and a monkey) and then calls some of their methods.

3) Do the following:
   i) In the class `Pet` of the previous exercise
      a) alter the fields `kind` and `legsNumber` to private fields `_kind` and `_legsNumber` correspondingly.
      b) add a getter and a setter method named `getKind` and `setKind` correspondingly. They will be used to get and set the value of the field `_kind`. The setter method must throw an error when the field is set to an empty value.
      c) add a getter and a setter method named `getLegsNumber` and `setLegsNumber` correspondingly. They will be used to get and set the value of the field `_legsNumber`. The setter method must throw an error when the field is set to a negative value.
      d) add a constructor to accept initial values for the private fields `_kind` and `_legsNumber` through its formal argument list.
   ii) Write a Java program that creates one instance of the class `Pets` (for example, a dog) and then calls both of its methods. Then try to set erroneous values for fields `kind` and `legsNumber` and see what happens.

4) Do the following:
   i) Write a class named `Box` that includes
      a) three private float (real) fields named `_width`, `_length`, and `_height`.
      b) a constructor that accepts initial values for the three fields `_width`, `_length`, and `_height` through its formal argument list.
      c) a public void method named `displayVolume` that calculates and displays the volume of a box whose dimensions are `_width`, `_length`, and `_height`. It is given that *volume* = *width* × *length* × *height*
      d) a public void method named `displayDimensions` that displays box's dimensions.

ii) Using the class cited above, write a Java program that prompts the user to enter the dimensions of 30 boxes, and then displays their dimensions and their volume.

Hint: Create an array of 30 objects of the class `Box`.

5) In the class `Box` of the previous exercise add three getter and three setter methods named `getWidth`, `getLength`, `getHeight`, and `setWidth`, `setLength`, `setHeight` correspondingly. They will be used to get and set the values of the fields `_width`, `_length`, and `_height`. The setter methods must throw an error when the corresponding field is set to a negative value or zero.

6) Do the following:

i) Write a class named `Cube` that includes

a) a private float (real) field named `_edge`.

b) a constructor that accepts an initial value for the field `_edge` through its formal argument list.

c) a public void method named `displayVolume` that calculates and displays the volume of a cube whose edge length is `_edge`. It is given that $volume = edge^3$

d) a public void method named `displayOneSurface` that calculates and displays the surface area of one side of a cube whose edge length is `_edge`.

e) a public void method named `displayTotalSurface` that calculates and displays the total surface area of a cube whose edge length is `_edge`. It is given that $total\ surface = 6 \times edge^2$

ii) Using the class cited above, write a Java

program that prompts the user to enter the edge length of a cube, and then displays its volume, the surface area of one of its sides, and its total surface area.

7) In the class `Cube` of the previous exercise add a getter and a setter method named `getEdge` and `setEdge` correspondingly. They will be used to get and set the value of the field `_edge`. The setter method must throw an error when the field is set to a negative value or zero.

8) Do the following:
   a) Write a class named `Circle` that includes
      i) a private float (real) field named `_radius` with an initial value of −1.
      b) a getter and a setter method named `getRadius` and `setRadius` correspondingly. They will be used to get and set the value of the field `_radius`. The

getter method must throw an error when the field has not yet been set, and the setter method must throw an error when the field is set to a negative value or zero.

c) a public method named `getDiameter` that calculates and returns the diameter of a circle whose radius is `_radius`. It is given that $diameter = 2 \times radius$

d) a public method named `getArea` that calculates and returns the area of a circle whose radius is `_radius`. It is given that $area = 3.14 \times radius^2$

e) a public

method named getPerimeter that calculates and returns

the perimeter of a circle whose radius is _radius. It is given that

perimeter = 2 × 3.14 × *radius*

iii) Write a subprogram name

d d i s p l a y M e n u th at di s pl a y s th e fo ll o w in g m e n u.

1) E nt er ra di

us
2) Displayradius
3) Displaydiameter
4) Displayarea
5) Displayperi

meter

6) Exiting the class and the subprogram cited above

o
v
e,
w
ri
te
a
Ja
v
a
pr
o
gr
a
m
th
at
di
s
pl
a
y
s
th
e
pr
e
vi
o
u
sl
y
m
e
nt
io
n

e
d
m
e
n
u
a
n
d
pr
o
m
pt
s
th
e
u
se
r
to
e
nt
er
a
c
h
oi
c
e
(o
f
1
to
6)
.
If
c

h
oi
c
e
1
is
se
le
ct
e
d,
th
e
pr
o
gr
a
m
m
u
st
pr
o
m
pt
th
e
u
se
r
to
e
nt
er
a
ra
di

u
s.
If
c
h
oi
c
e
2
is
se
le
ct
e
d,
th
e
pr
o
gr
a
m
m
u
st
di
s
pl
a
y
th
e
ra
di
u
s
e

nt
er
e
d
in
c
h
oi
c
e
1.
If
c
h
oi
c
es
3,
4,
or
5
ar
e
se
le
ct
e
d,
th
e
pr
o
gr
a
m
m
u

st
di
s
pl
a
y
th
e
di
a
m
et
er
,
th
e
ar
e
a,
or
th
e
p
er
i
m
et
er
c
or
re
s
p
o
n
di
n

gl
y
of
a
ci
rc
le
w
h
o
se
ra
di
u
s
is
e
q
u
al
to
th
e
ra
di
u
s
e
nt
er
e
d
in
c
h
oi
c

e

1. The process must repeat as many times as the user wishes.

9) Assume that you work in a computer software company that is going to create a word processor application. You are assigned to write a class that will be used to provide information to the user.

   i) Write a class named `Info` that includes a) a private string field named `_userText`.

b) a getter and a setter method named getUserText and s

etUserTextcorrespondingly. They will be used to get an

d
se
t
th
e
v
al
u
e
of
th
e
fi
el
d
_
u
s
e
r
T
e
x
t.
T
h
e
se
tt
er
m
et
h
o
d
m
u

st
th
ro
w
a
n
er
ro
r
w
h
e
n
th
e
fi
el
d
is
se
t
to
a
n
e
m
pt
y
v
al
u
e.

c) a
p
u
bl
ic

method named getSpaces countThatReturnsTheTotalNu

mber of spaces that field_user Text contains.

d) a public meth

odnamedgetwordscountthatreturnsthetotalnumberof

words that field_user Text contains.

e) a public method name

edgetvowelscountthatreturnsthetotalnumberofvowel

s that field_user_Text contains.

f) a public method named g

```
etLettersCountthatreturnsthetotalnumberofcharacter
```

s (excludingspaces) that field_user_Text contains.

ii) Using the class

cited above, write a testing program that prompts the user to enter a text and then displays all available information. Assume that the user enters only space characters or letters (upper case or lower

case) and the words are separated by a single space character.

Hint: In a text of three words, there are two spaces, which means that the total number of words is one more than the total number of spaces. Count the total number of spaces, and then you can easily find the total

number of words!

10) During the Cold War after World War II, messages were encrypted so that if the enemies intercepted them, they could not decrypt them without the decryption key. A very simple encryption algorithm is alphabetic rotation. The algorithm moves all letters N steps "up" in the alphabet,

where N is the encryption key. For example, if the encryption key is 2, you can encrypt a message by replacing the letter A with the letter C, the letter B with the letter D, the letter C with the letter E, and so on. Do the following:

i) Write a class named `EncryptDecrypt` that includes a) a private integer field named `_encrDecrKey` with

an initial value of −1.

b) a getter and a setter method named getEncrDecrK

eyandsetEncrDecrKeycorresponding
y. They will b

e
u
se
d
to
g
et
a
n
d
se
t
th
e
v
al
u
e
of
th
e
fi
el
d
_
e
n
c
r
D
e
c
r
K
e
y.
T

h
e
g
et
te
r
m
et
h
o
d
m
u
st
th
ro
w
a
n
er
ro
r
w
h
e
n
th
e
fi
el
d
h
as
n
ot
y
et

b
e
e
n
se
t,
a
n
d
th
e
se
tt
er
m
et
h
o
d
m
u
st
th
ro
w
a
n
er
ro
r
w
h
e
n
th
e
fi

eld is not set to a value between 1 and 26.

c) a public method na

m
e
d
e
n
c
r
y
p
t
th
at
a
c
c
e
pt
s
a
m
es
sa
g
e
th
ro
u
g
h
it
s
fo
r
m
al
ar
g

u
m
e
nt
li
st
a
n
d
th
e
n
re
tu
rn
s
th
e
e
n
cr
y
pt
e
d
m
es
sa
g
e.

d) a
p
u
bl
ic
m
et

hodnamedecryptthatacceptsanencryptedmessage

through its formal argument list and then returns the decrypte

d
m
es
sa
g
e.

ii) Write a subprogram named `displayMenu` that displays the following menu:

1) Enter encryption/decryption key
2) Encrypt a message
3) Decrypt a message
4) Exit

iii) Using the class

and the subprogram cited above, write a Java program that displays the menu previously mentioned and then prompts the user to enter a choice (of 1 to 4). If choice 1 is selected, the program must promp

t the user to enter an encryption/decryption key. If choice 2 is selected, the program must prompt the user to enter a message and then display the encrypted message. If choice 3 is selected, the progra

m must prompt the user to enter an encrypted message and then display the decrypted message. The process must repeat as many times as the user wishes. Assume that the user enters only

lower case letters or a space for the messa ge.

11) Do the following:

i) Write a parent class named `Vehicle` that includes a) a public integer field named `numberOfWheels`, a public string field named `color` and three public float (real) fields named `length`, `width`, and `height`.

b) a c

onstructorthatacceptsinitialvaluesforthefieldsnumberof

wheels, color, length, width, and height through it

s
fo
r
m
al
ar
g
u
m
e
nt
li
st
.

c) tw
o
p
u
bl
ic
v
oi
d
m
et
h
o
d
s
n
a
m
e
d
s
t
a

rtEnginestopEnginethatdisplaythemessages“T

h
e
e
n
gi
n
e
st
ar
te
d
”
a
n
d
“
T
h
e
e
n
gi
n
e
st
o
p
p
e
d
”,
c
or
re
s
p
o

n
di
n
gl
y.

ii) Write a child class named `Car` that inherits the class `Vehicle`. Additionally, it includes a) a constructor with an additional public integer field named `bootCapacity` and an initial

value of zero.

b) a public void methodnamedturnWindshieldWi

person that displays the message “The windshield wipersh

a v e b e e n tu rn e d o n! ”.

iii) Write a child class named `Motorcycle` that inherits the class `Vehicle`. Additionally, it must includ

a) e a constructor with an

additional public Boolean field named `hasLuggage` and an initial value of `false`.

b) a public void method named `doAWh`

eelie that displays the message “I amdoing a wheelie!!!”

iv) Using the

classes cited above, write a Java program that creates two instances of the class `Car` and one instance of the class `Motorcycle`, assigns some values to their fields, and then calls all of their methods.

12) Alter the Java program of Section 39.7 – Class Inheritance (`Project_39.7`) as follows:

i) In the class `SchoolMember`, alter the fields `name` and `age` to private fields `_name` and `_age` correspondingly, and add getter and setter methods for both of them. The setter method of the field `_name` must throw an error when it is set to

an empty value, whereas the setter method of the field `_age` must throw an error when it is set to a negative value or zero.

ii) In the class `Teacher`, alter the field `salary` to private field `_salary`, and add getter and setter methods for it. The setter method

must throw an error when the field is set to a negative value.

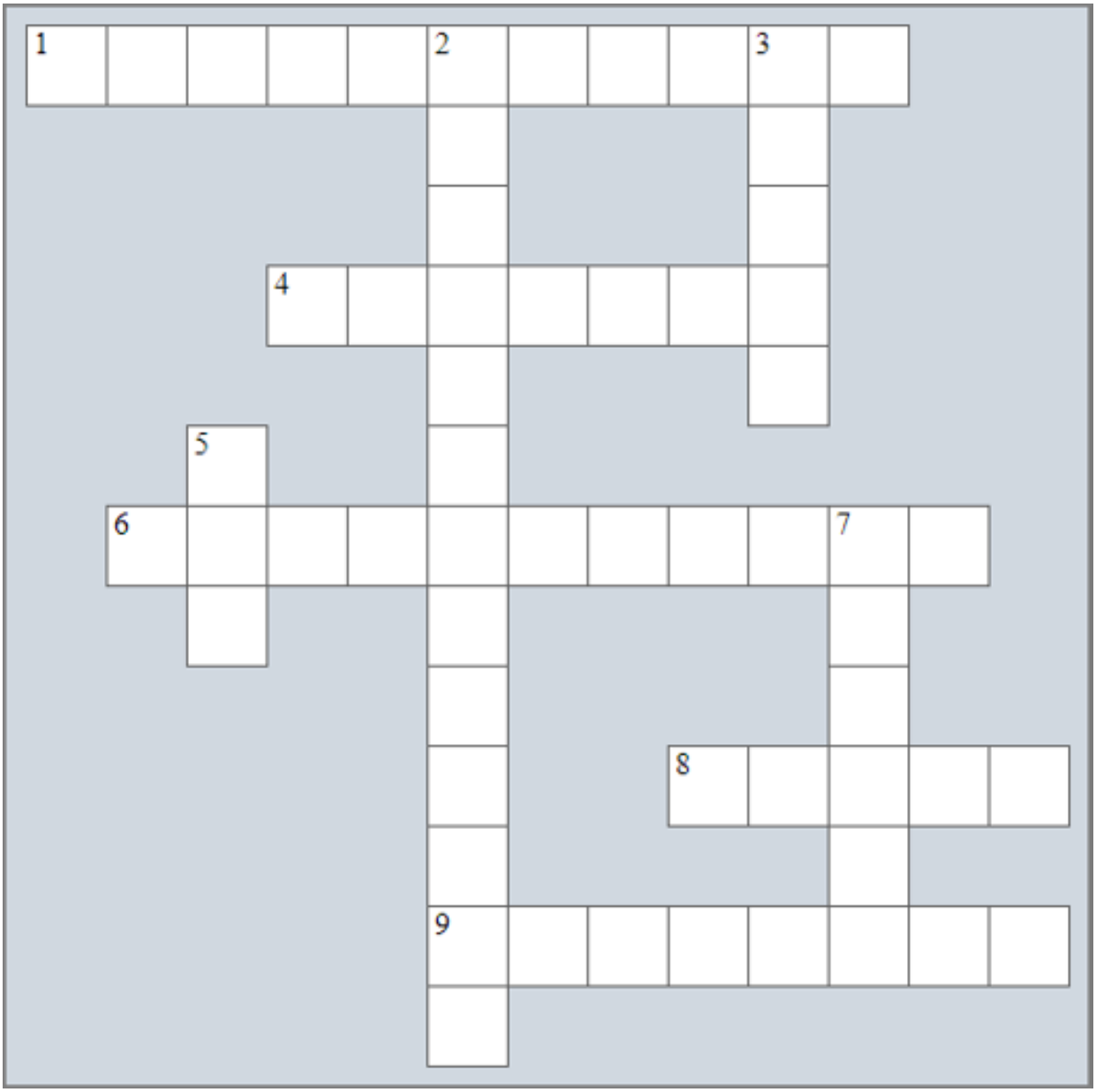
iii) In the class `Student`, alter the field `finalGrade` to private field `_finalGrade`, and add getter and setter methods for it. The setter method

must throw an error when the field is set to a value other than A, B, C, D, E, or F.

Review in “Object-Oriented Programming”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

1) Class _____ lets you write a class using another class as a base.

4) The actions that an object performs.

6) This method is executed automatically whenever an object is created.

8) An object's attribute.

9) Object-_____ programming is a style of programming that focuses on objects.

Down

2) The process of creating a new object.

3) Every object is created from a _____.

5) In _____ you can combine data and functionality and enclose them inside something called an object.

7) A class instance.

Review Questions

Answer the following questions.

1) What is object-oriented programming?
2) What is the constructor of a class?
3) When do you have to write a field name using dot notation?
4) What is the `this` keyword?
5) Why a field should not be exposed in OOP?
6) What are the getter and setter methods?
7) What is meant by the term “class inheritance”?

Part IX
Files

Chapter 40
Introduction to Files

40.1 Introduction

All programs you have seen so far can be described as “temporary”. Even though they read some input data and display some output results, all of the values are stored in variables, arrays, and other data structures that exist in the main memory (RAM) of your computer; therefore, these values are all lost when the program finishes executing. Even if this doesn't happen, they are certainly lost when you shut down your computer. There are many cases, however, where you need to keep those values in more permanent storage devices, such as a Hard Disk Drive (HDD) or a Solid State Drive (SSD).

Java can read input data stored in a computer file or write output results in the same or a different file. This reading/writing process is called *File I/O* (*File Input*/*Output*) and can be implemented with some of Java's ready-to-use methods.

Usually, the type of file being used is a *text file*. A text file contains a sequence of characters and is stored in a permanent storage device (HDD, SSD etc.).

✎ *Another type of file being used in computer programming is a “binary file”; however, this type is beyond the scope of this book and will be no further analyzed.*

In the following sections, you will learn how to open and close a text file, how to read values from or write values in a text file, and even how to search a value within it.

40.2 Opening a File

The `Scanner` and `FileWriter` are two classes that can be used for reading data from and writing data in a text file correspondingly. The `FileWriter` class can also be used to append data to a text file.

To use a file for reading, the first thing you need to do is open the file. In Java, this is accomplished using the following statements given in general form

Scanner *descriptor*; *descriptor* = new **Scanner**(Paths**.get**(*filename*));

or the more concise statement

Scanner *descriptor* = new **Scanner**(Paths**.get**(*filename*));

where

- `descriptor` is the name of a file object and can be used to read from a file.
- `filename` is a string that contains the folder (directory) and the name of the file stored in the hard disk (or any other storage device, such as SSD, Flash USB disk etc.). If the file `filename` does not exist, Java throws a runtime error.

> ✎ *When you open a file for reading, the file pointer is positioned at the beginning of the file (position 0).*
>
> ✎ *To some extent, the file pointer can be likened to the index of an array. You will learn more about the file pointer in Section 40.5.*

Correspondingly, to use a file for writing, the first thing you need to do is open the file using the following statements given in general form

FileWriter *descriptor*; *descriptor* = new **FileWriter**(*filename*);

or the more concise statement

FileWriter *descriptor* = new **FileWriter**(*filename*);

If the file `filename` already exists, Java overwrites it; otherwise, Java creates a new file.

Similarly, to use a file for appending, you need to open the file using the following statements given in general form

FileWriter *descriptor*; *descriptor* = new **FileWriter**(*filename*, true);

or the more concise statement

FileWriter *descriptor* = new **FileWriter**(*filename*, true);

In this case, if the file `filename` does not exist, Java creates a new file.

> ✎ *When you open a file for appending, the file pointer is positioned at the end of the file.*

Let's see some examples.

Example 1

The following statement

Scanner f = new **Scanner**(Paths**.get**("*names.txt*"));

opens the text file “names.txt” for reading. The file “names.txt” must exist in the same folder (directory) where the executable file (.class) has been saved. If the file does not exist, Java throws a runtime error.

Example 2

The statement

FileWriter fgrades = new **FileWriter**("*c:/temp/grades.txt*");

creates the text file “grades.txt” in the folder (directory) “c:/temp” and opens it for writing. If the file already exists, Java overwrites it.

> ✎ *Note that the path definition of a file uses the slash (/) and not the backslash (\ ) character.*

Example 3

The statement

FileWriter fgrades = new **FileWriter**("*c:/temp/students/grades.txt*", true);

opens the text file “grades.txt” for appending. The file must exist in the subfolder (subdirectory) “students” of the folder (directory) “c:/temp”. If the file does not exist, Java creates a new file.

> ✎ *In order to use the* `Scanner`, `Paths` *and* `FileWriter` *classes, you need to import the corresponding libraries. Also, every method or void method (including the void method* `main`*) that uses these classes must throw exceptions, as shown in the example that follows.*
>
> ```
> import java.util.Scanner; import java.nio.file.Paths; import java.io.FileWriter;
> public class App {
>    public static void main(String[] args) throws Exception {
>       Scanner f = new Scanner(Paths.get("a_file.txt"));
>       FileWriter fout = new FileWriter(PATH + "b_file.txt");
>       …
>       …
>    }
> }
> ```

40.3 Closing a File

After completing reading, writing, or appending operations on a file, it is crucial to close the file using the `close()` method. This method states that the use of the file has been completed, leading the operating system (OS) to

save any unsaved data that may exist in the main memory (RAM). The general form of the `close()` method is as follows:

descriptor.**close**();

where *`descriptor`* is the name of the file object that was used to open the file.

Let's see some examples.

Example 1

The following code fragment

Scanner fst = new **Scanner**(Paths.**get**("*c:/temp/data.txt*"));

```
A statement or block of statements
```

fst.**close**();

opens the text file “c:/temp/data.txt” for reading and, at the end, it closes it.

Example 2

The following code fragment

FileWriter f = new **FileWriter**("*temperatures.txt*", true);

```
A statement or block of statements
```

f.**close**();

opens the text file “temperatures.txt” for appending and, at the end, it closes it.

40.4 Writing in (or Appending to) a File

To write a string in (or even append a string to) a file, you can use the `write()` method. The general form of this method is as follows:

descriptor.**write**(*str*);

where

- *`descriptor`* is the name of the file object that was used to open the file.
- *`str`* is the string value that you want to write in (or append to) the file.

The following example creates the file “f_data40.4-i.txt” in the folder (directory) “c:/temp”. If the file “f_data40.4-i.txt” already exists, Java overwrites it; otherwise, Java creates a new file. Then, the program writes three strings in the file, using the `write()` method.

Project_40.4a

```
static final String PATH = "c:/temp/";
```

```
public static void main(String[] args) throws Exception {
    FileWriter fout = new FileWriter(PATH + "f_data40.4-
                          i.txt");
        fout.write("Good Morning"); fout.write("Good
            Evening"); fout.write("Good Night");
                        fout.close(); }
```

Try to execute the above program and then locate and open (using a notepad application) the recently created “c:/temp/f_data40.4-i.txt” file. What you see in the file is the following:

Good MorningGood EveningGood Night

All three strings were written in a single row. This happened because, unlike the `println()` method that you're familiar with, the `write()` method does not automatically add a “line break” at the end of the string.

> ✎ *To open a text file and see what is written inside, you can use a simple notepad application, such as the Notepad of Windows. Alternatively, you can download free of charge and use the Notepad++ application, from the following address: https://notepad-plus-plus.org*

To force Java to write a “line break”, you can use the special sequence of characters `\n` (presented in Section 6.2). The next example opens the previously created file “c:/temp/f_data40.4-i.txt” for appending. Subsequently, a “line break” is written along with three lines of text.

```
                    🗀 Project_40.4b
          static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
    FileWriter fout = new FileWriter(PATH + "f_data40.4-
                          i.txt", true);
       fout.write("\n"); fout.write("Good Morning\n");
        fout.write("Good Evening\n"); fout.write("Good
                          Night\n");
                        fout.close(); }
```

If you execute this program, and then locate and open the “c:/temp/f_data40.4-i.txt” file with a notepad application, you will now see the following content:

Good MorningGood EveningGood Night Hello!

Hi!
Bye!

> ✎ *The first line "Good MorningGood EveningGood Night" was already in the file before opening the file for appending.*

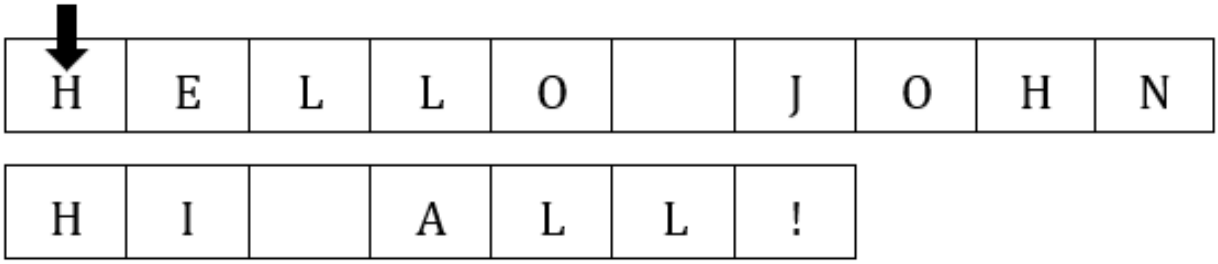
The next example creates the file "f_data40.4-ii.txt" in the folder "c:/temp". If the file "f_data40.4-ii.txt" already exists, Java overwrites it, otherwise, Java creates a new file. Then, the program writes 10 strings on 10 separate lines in the file.

```
                    Project_40.4c
          static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
       int i; FileWriter fout = new FileWriter(PATH +
                  "f_data40.4-ii.txt", true);
                  for (i = 1; i <= 10; i++) {
                 fout.write("Line " + i + "\n");
                              }
                          fout.close(); }
```

40.5 The File Pointer

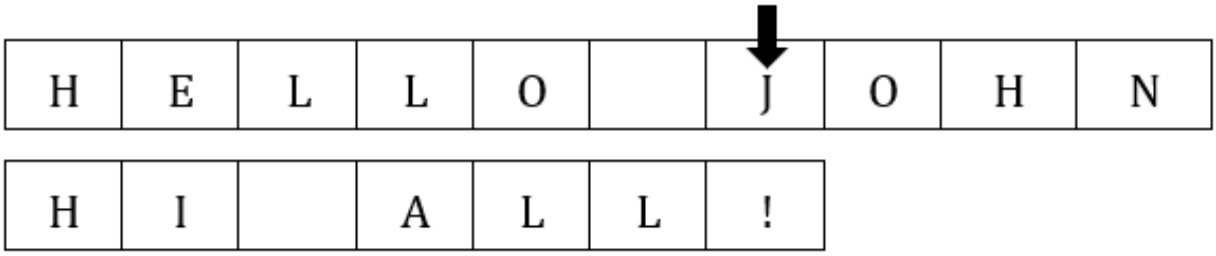
As already mentioned, the file pointer is quite similar to the index of an array. Both are used to specify the point from which to read information or where to write new information. However, the main distinction between the file pointer and the array index is that the former is automatically moved every time a read or write operation is performed.

Let's assume a file already contains the messages "HELLO JOHN\nHI ALL!". If you open this specific file for reading, the file pointer is automatically placed at the beginning of the file, as shown below.



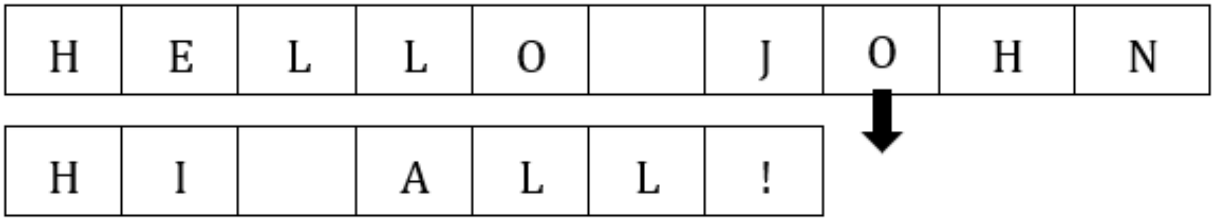
If you now perform a read operation (as described in the next section), the reading will commence from the position indicated by the file pointer, and

the pointer will automatically advance towards the end, moving as many positions as the characters you have read. Below is the position the file pointer will be in if you read one line from the file.

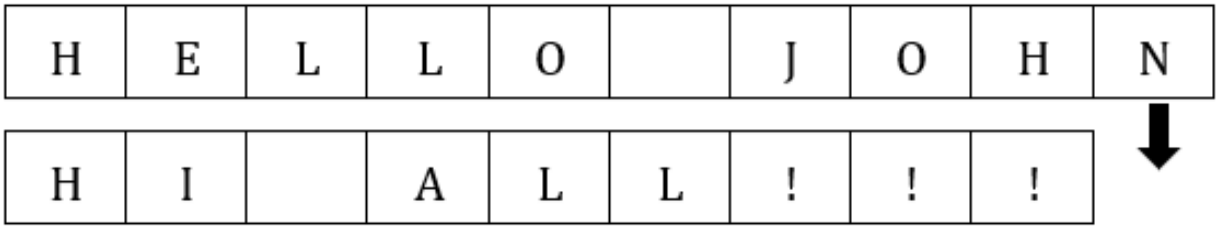


✎ *The subsequent read operation will start from the position where the word "HI" begins.*

On the contrary, if you open a file for appending, the file pointer is automatically positioned at the end of the file, as illustrated here:



If you then perform a write operation, the writing will commence from the position indicated by the file pointer, and the pointer will automatically advance towards the end, moving as many positions as the characters you have written in the file.



40.6 Reading from a File

Suppose the file "f_data40.6.txt" contains the following text.

```
15.3 15 Round the number down 22.6 23 Round the number up 55.5 56 Round the number up again
```

Reading until the end of the current line To read all the characters from the current position of a file until the end of the current line and assign them to variable *`var_name_str`*, you can use the following statement given in general form

var_name_str = *descriptor*.**nextLine**()

where

- *descriptor* is the name of the file object that was used to open the file.
- *var_name_str* can be any variable of type `String`.

The program that follows assigns the three lines of the file "f_data40.6.txt" to the variables `s1`, `s2`, and `s3`.

```
Project_40.6a
        static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
            String s1, s2, s3; Scanner fin = new
         Scanner(Paths.get(PATH + "f_data40.6.txt"));
        s1 = fin.nextLine(); s2 = fin.nextLine(); s3 =
                        fin.nextLine();
                         fin.close();
    System.out.println(s1); //It displays: 15.3 15 Round
   the number down System.out.println(s2); //It displays:
     22.6 23 Round the number up System.out.println(s3);
     //It displays: 55.5 56 Round the number up again }
```

Reading a value To read a value from the current position of the file until the first space character or until the end of the current line (whichever comes first) and assign it to a variable, you can use one of the following statements given in general form:

var_name_int = *descriptor*.**nextInt**(); *var_name_dbl* = *descriptor*.**nextDouble**();

where

- *descriptor* is the name of the file object that was used to open the file.
- *var_name_int* can be any variable of type `int`.
- *var_name_dbl* can be any variable of type `double`.

The following program assigns the first two values from the file "f_data40.6.txt" to the variables `d1`, `i1` and the text of the first line to the variable `s1`. Then, it perform the same operation for the second line of the file, assigning the values to the variables `d2`, `i2` and `s2` respectively.

```
Project_40.6b
        static final String PATH = "c:/temp/";
```

```
public static void main(String[] args) throws Exception {
    int i1, i2; double d1, d2; String s1, s2; Scanner fin
     = new Scanner(Paths.get(PATH + "f_data40.6.txt"));
      d1 = fin.nextDouble(); i1 = fin.nextInt(); s1 =
        fin.nextLine(); d2 = fin.nextDouble(); i2 =
            fin.nextInt(); s2 = fin.nextLine();
                        fin.close();
        System.out.println(d1); //It displays: 15.3
         System.out.println(i1); //It displays: 15
      System.out.println(s1); //It displays: Round the
     number down System.out.println(d2); //It displays:
                            22.6
         System.out.println(i2); //It displays: 23
      System.out.println(s2); //It displays: Round the
                        number up }
```

The `nextLine()` *method reads all the characters from the current position of a file until the end of the current line.*

In the US, the decimal separator is a period (.). In many countries, however, the decimal separator is a comma (,), not a period (.). For example, if you live in Europe and attempt to execute this code, a runtime error will occur. To resolve this, change in the file "f_data40.6.txt" the values 15.3, 22.6, and 55.5 to 15,3, 22,6, and 55,5, respectively.

40.7 Iterating Through the Contents of a File

You can iterate through the contents of a file using a while-loop.

Let's suppose the file "f_to_be_or_not_to_be.txt" contains the following text:

To be, or not to be: that is the question: Whether 'tis nobler in the mind to suffer The slings and arrows of outrageous fortune, Or to take arms against a sea of troubles, And by opposing end them? To die: to sleep;

The following example displays all the lines of the file, one at each iteration.

```
                    Project_40.7a
        static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
```

```
String line; Scanner fin = new Scanner(Paths.get(PATH
            + "f_to_be_or_not_to_be.txt"));
              while (fin.hasNextLine()) {
                  line = fin.nextLine();
                System.out.println(line);
                              }
                          fin.close(); }
```

✎ *The* `hasNextLine()` *method returns* `false` *when the current position is at the end of the file.*

Suppose the file "f_data40.7.txt" contains the following values.

```
15 19
22 28
55 50
```

The following example displays all six values from the file, one at each iteration.

```
                    Project_40.7b
        static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
     int x; Scanner fin = new Scanner(Paths.get(PATH +
                    "f_data40.7.txt"));
      while (fin.hasNextInt()) { //Use hasNextDouble()
         method when reading real numbers (floats)!
                      x = fin.nextInt();
                    System.out.println(x);
                              }
                          fin.close(); }
```

✎ *The* `hasNextInt()` *and* `hasNextDouble()` *methods return* `false` *when the current position is at the end of the file.*

40.8 Review Questions: True/False

Choose **true** or **false** for each of the following statements.

1) The contents of a file are lost when you shut down your computer.
2) If you open a file using the `Scanner f = new Scanner(Paths.get(`*`filename`*`))` statement and the file *`filename`* does not

exist, Java creates a new one.

3) If you open a file using the `FileWriter f = new FileWriter(`*`filename`*`, true)` statement and the file *`filename`* does not exist, Java creates a new one.

4) If you open a file using the `FileWriter f = new FileWriter(`*`filename`*`)` statement and the file *`filename`* does not exist, Java throws a runtime error.

5) The statement `FileWriter f = new FileWriter(`*`filename`*`, true)` overwrites the file *`filename`* (when the file already exists).

6) The following program is correct
public static void main(**String**[] args) throws Exception {
Scanner ff = new **Scanner**(Paths.**get**("*grades.txt*"));
System.out.**println**(ff.**nextLine**()); fff.**close**(); }

7) The following code fragment is correct
FileWriter f = new **FileWriter**("*grades.txt*");
System.out.**println**(f.**nextLine**());

8) The following code fragment is correct
Scanner f = new **FileWriter**("*grades.txt*"); f.**write**("*OK*");

9) The following code fragment is correct
FileWriter f = new **FileWriter**("*grades.txt*"); f.**write**("*OK*");

10) If there are 10 characters in a file named “test.txt”, after executing the following program, the size of the file gets bigger.

```
public static void main(String[] args) throws Exception {
   FileWriter f = new FileWriter("test.txt"); f.write("Hello"); f.close(); }
```

11) If there are 10 characters in a file named “test.txt”, after executing the following program, the size of the file gets bigger.

```
public static void main(String[] args) throws Exception {
   FileWriter f = new FileWriter("test.txt", true); f.write("Hello"); f.close(); }
```

12) The following code fragment is correct.

```
FileWriter f = new FileWriter("c:\data\test.txt"); f.write(10); f.close();
```

13) After repeatedly executing the following program three times, there will be only two lines of text in the file “test.txt”.

```
public static void main(String[] args) throws Exception {
   FileWriter f = new FileWriter("test.txt", true); f.write("Good Morning\n");
   f.write("Good Evening\n"); f.close(); }
```

14) After repeatedly executing the following program three times, there will be only two lines of text in the file “test.txt”.

```
public static void main(String[] args) throws Exception {
   FileWriter f = new FileWriter("test.txt"); f.write("Good Morning");
   f.write("Good Evening"); f.close(); }
```

15) After repeatedly executing the following program three times, there will be only two lines of text in the file “test.txt”.

```
public static void main(String[] args) throws Exception {
   FileWriter f = new FileWriter("test.txt"); f.write("Good Morning\n");
   f.write("Good Evening\n"); f.close(); }
```

16) The `nextLine()` method reads one line from a file.

17) The `nextLine()` method always reads all the characters from the current line.

18) You cannot use a while-loop to iterate through the contents of a file.

19) You cannot use a for-loop to iterate through the contents of a file.

20) Suppose there are two lines of text in a file named “test.txt”. After executing the following code fragment, only one line of text will be displayed on the user's screen.

```
Scanner fin = new Scanner(Paths.get("test.txt")); while (fin.hasNextLine()) {
   System.out.print(fin.nextLine()); }
fin.close();
```

21) If the current position is at the end of a file, the `hasNextLine()` method returns `false`.

22) If the file “test.txt” contains the text as shown below
Hello
World!

then, the following code fragment displays “LOL!” without the double quotes on the screen.

```
Scanner f = new Scanner(Paths.get("test.txt")); String[] x = new String[2]; x[0] =
f.nextLine(); x[1] = f.nextLine(); f.close();
String a = ""; a += x[0].charAt(2); a += x[0].charAt(4); a += x[1].charAt(3); a +=
x[1].charAt(5);
System.out.println(a.toUpperCase());
```

40.9 Review Exercises

Complete the following exercises.

1) Write a Java program that creates a text file and writes the days of the week (Sunday, Monday etc.), one on each line.
2) Write a Java program that reads the days of the week from the file created in the previous exercise (Sunday, Monday etc.) and stores them into an array. Then, the program must display the days of the week in the exact reverse of the order in which they are stored in the array.
3) Write a Java program that appends to the file of the previous exercise the text "*** End of File ***", without the double quotes.
4) Write a Java program that writes 50 random integers (between 1 and 100) in a file named "randoms.txt", one on each line.
5) Write a Java program that creates 10 files named "file1.txt", "file2.txt", … "file10.txt" and writes a random 3-digit integer in each one.
6) Write a Java program that writes the following multiplication table in a file.

   1 x 1 = 1

   1 x 2 = 2

   1 x 3 = 3

   1 x 4 = 4

   2 x 1 = 2

   2 x 2 = 4

   2 x 3 = 6

   2 x 4 = 8

   …

   …

   10 x 1 = 10

   10 x 2 = 20

   10 x 3 = 30

   10 x 4 = 40
7) Write a Java program that displays the number of characters that exist in each line of a file.
8) Write a Java program that, for each line of a file, displays the message "There is a punctuation mark on line No XX", in case there is a punctuation mark in the line (check only for commas, periods, and

exclamation marks). Please note that XX must be replaced by an actual value.

Chapter 41
More with Files

41.1 Exercises of a General Nature with Files

Exercise 41.1-1 Calculating the Sum of 10 Numbers

Suppose there is a file named "f_data41.1-1.txt" that contains 10 3-digit integers (separated by a single space character). An example of the structure of the file is shown here.

```
131 500 122 152 127 191 111 290 156 161
```

Write a Java program that calculates and displays their sum.

Solution There are two approaches to iterate through the numbers of this file.

First approach In this approach, a string variable named `values` gets the content of the whole line of the file. Then, the `substring()` method is used in a for-loop to split the content into individual three-digit numbers. These numbers are then converted to integers using the `Integer.parseInt()` method to calculate their sum.

```
Project_41.1-1a
static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception
{
int i, total; String values;
Scanner fin = new Scanner(Paths.get(PATH +
"f_data41.1-1.txt")); values = fin.nextLine();
fin.close();
total = 0; for (i = 0; i < 10; i++) {
total += Integer.parseInt(values.substring(i * 4,
i * 4 + 3));
}
System.out.println(total); }
```

Second approach Since the values are separated by a single space character, you can use the `nextInt()` method in a for-loop. The `nextInt()` method reads one value at each iteration.

Project_41.1-1b

```
static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception
{
int i, total;
Scanner fin = new Scanner(Paths.get(PATH +
"f_data41.1-1.txt")); total = 0; for (i = 0; i < 10;
i++) {
total += fin.nextInt();
}
fin.close();
System.out.println(total); }
```

Exercise 41.1-2 Calculating the Average Value of an Unknown Quantity of Numbers

Suppose there is a file named "f_data41.1-2.txt" that contains numbers, one on each line, except the last one which contains the phrase "End of file". An example of the structure of the file is shown here.

```
16
13.172
33.5
.
.
End of file
```

Write a Java program that calculates and displays their average value.

Solution According to the "Ultimate" rule discussed in Section 29.3, the while-loop should be as follows, given in general form.

```
sNumber = fin.nextLine(); while (!sNumber.equals("End of file")) {
    A statement or block of statements
    sNumber = fin.nextLine(); }
```

The final program is as follows.

Project_41.1-2

```
static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
   double total = 0; int count = 0; String sNumber; Scanner fin = new
   Scanner(Paths.get(PATH + "f_data41.1-2.txt"));
```

```
   sNumber = fin.nextLine(); while (!sNumber.equals("End of file")) {
      total += Double.parseDouble(sNumber);
      count += 1;
      sNumber = fin.nextLine();
   }
   fin.close();
   if (count > 0) {
      System.out.println(total / count);
   }
}
```

Exercise 41.1-3 Finding Minimum and Maximum Values

Suppose there is a file named "f_data41.1-3.txt" that contains numbers, one on each line. An example of the structure of the file is shown here.

```
16
13.172
33.5
.
.
```

Write a Java program that finds the greatest and lowest values and stores them in a file named "output.txt" in the following form

33.5 13.172

Assume that there is at least one value in the file "f_data41.1-3.txt".

Solution The final program is as follows.

Project_41.1-3

```
static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
   double number, maximum, minimum; Scanner fin = new Scanner(Paths.get(PATH +
   "f_data41.1-3.txt"));
   //Read the first value maximum = fin.nextDouble(); minimum = maximum;
   //Read the rest of the values, line by line while (fin.hasNextDouble()) {
      number = fin.nextDouble();
      if (number > maximum ) {
         maximum = number;
      }
      if (number < minimum) {
         minimum = number;
      }
   }
   fin.close();
```

```
//Write the greatest and lowest value in a file FileWriter fout = new
FileWriter(PATH + "output.txt"); fout.write(maximum + " " + minimum + "\n");
fout.close(); }
```

Exercise 41.1-4 Concatenating Files

Suppose there are two text files named "text1.txt" and "text2.txt". Write a Java program that concatenates the contents of the two files and writes the concatenated text in a third file named "final.txt", placing the contents of the file "text1.txt" before the contents of the file "text2.txt".

Solution This exercise can be solved using several approaches. Let's see two of them.

First approach The program opens the file "text1.txt", reads all of its contents, assigns them to the variable `contents`, and then closes the file. Next, it opens the file "text2.txt", reads all of its contents, concatenates them with the previous ones (those in the variable `contents`), and closes the file. Finally, it opens the file "final.txt" and writes the concatenated contents in it, as shown in the program that follows.

```
Project_41.1-4a
static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception
{
String contents = ""; Scanner fin;
fin = new Scanner(Paths.get(PATH + "text1.txt"));
while (fin.hasNextLine()) {
contents += fin.nextLine() + "\n";
}
fin.close();
fin = new Scanner(Paths.get(PATH + "text2.txt"));
while (fin.hasNextLine()) {
contents += fin.nextLine() + "\n";
}
fin.close();
FileWriter fout = new FileWriter(PATH +
"final.txt"); fout.write(contents); fout.close(); }
```

✎ *Note how the object* `fin` *is declared at the beginning of the program. This way you can use the same object again and again to open and read multiple files.*

Second approach The program that follows opens the three files, writes the concatenated contents, and then closes them.

```
                🗀 Project_41.1-4b
        static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception
                              {
        String contents = ""; Scanner fin1 = new
   Scanner(Paths.get(PATH + "text1.txt")); Scanner fin2
      = new Scanner(Paths.get(PATH + "text2.txt"));
         FileWriter fout = new FileWriter(PATH +
                        "final.txt");
              while (fin1.hasNextLine()) {
            contents += fin1.nextLine() + "\n";
                              }
              while (fin2.hasNextLine()) {
            contents += fin2.nextLine() + "\n";
                              }
                  fout.write(contents);
      fout.close(); fin2.close(); fin1.close(); }
```

✎ *The order in which you close the files doesn't need to correspond to the order in which you opened them. You can open, for example, the file "text2.txt" first, and close it last, or open it first and close it first.*

Exercise 41.1-5 Searching in a File

In the United States, the Social Security Number (SSN) is a nine-digit identity number applied to all U.S. citizens in order to identify them for the purposes of Social Security. Suppose there is a file named "SSN.txt" that contains the SSNs (Social Security Numbers) of all citizens of the state of California, as well as their full name, one record on each line. An example of the structure of the file is shown here.

```
123456789 Aristides Bouras 123000000 Loukia Ainarozidou 121212121 John Papas Junior .
```

.

Write a Java program that prompts the user to enter the first digits of an SSN to search and then searches in the file and displays the first and last names of all citizens whose SSN starts with those digits.

Solution In this particular exercise, if the user enters all nine digits of an SSN to search, and this SSN is found, the program must display the first and last name of the corresponding citizen and stop searching thereafter. On the other hand, if the user enters less than nine digits to search, the program must search and display the first and last name of all the citizens whose SSN starts with those digits. The solution to this exercise is as follows.

Project_41.1-5

```
static final String PATH = "c:/temp/";
public static void main(String[] args) throws Exception {
   String ssnToSearch, line, ssn, fullName; boolean found;
   System.out.print("Enter an SSN to search: "); ssnToSearch = cin.nextLine();
   Scanner fin = new Scanner(Paths.get(PATH + "SSN.txt"));
   found = false; while (fin.hasNextLine()) {
      line = fin.nextLine();
      ssn = line.substring(0, 9);
      fullName = line.substring(10);
      //If it is found
      if (ssnToSearch.equals(ssn.substring(0, ssnToSearch.length()))) {
         System.out.println(fullName);
         found = true;
         //If SSN to search contains 9 digits and it is found, exit loop
         if (ssnToSearch.length() == 9) {
            break;
         }
      }
   }
   fin.close();
   if (!found) {
      System.out.println("Not found!");
   }
}
```

Exercise 41.1-6 Combining Files with Subprograms

Do the following: i) Write a subprogram named `readContent` *that accepts the filename of a file and returns its contents.*

ii) *Write a subprogram named* `writeContent` *that accepts the filename of a file and a string, and writes that string in the file.*

iii) *Using the subprograms cited above, write a Java program that prompts the user to enter the filenames of two files and then copies the contents of the first file to the second one. Assume that the user enters valid filenames.*

Solution Even though this particular exercise seems quite easy, it is necessary to highlight some things about the `readContent()` method. Examine the `readContent()` method that follows and try to find the error it contains. The error is a logic error, not a syntax one. This method is executed successfully without any syntax errors thrown by the Java interpreter. However, the error is there and quite difficult to spot!

```
static String readContent(String filename) throws Exception {
   Scanner fin = new Scanner(Paths.get(filename));
   String contents = ""; while (fin.hasNextLine()) {
      contents += fin.nextLine() + "\n";
   }
   return contents;
   fin.close(); }
```

What happens here is that, when the `return` statement is executed, Java ignores the rest of the statements of the method, which means that the file probably won't close. Imaging calling this particular method many times in a program. You would end up with many open files that will probably never close.

✎ *Latest versions of Java will probably close the file automatically when there is no reference* (`fin`) *to it, but it is bad practice to leave a file open and wait for Java to close it for you.*

A good practice is to place the `return` statement at the end of the method, as shown in the following program.

Project_41.1-6

```
static String readContent(String filename) throws Exception {
   Scanner fin = new Scanner(Paths.get(filename));
   String contents = ""; while (fin.hasNextLine()) {
      contents += fin.nextLine() + "\n";
   }
   fin.close();
```

```
   return contents; }
static void writeContent(String filename, String contents) throws Exception {
   FileWriter fout = new FileWriter(filename); fout.write(contents); fout.close(); }
public static void main(String[] args) throws Exception {
   System.out.print("Enter source filename: "); String source = cin.nextLine();
   System.out.print("Enter destination filename: "); String destination =
   cin.nextLine();
   String c = readContent(source); //Equivalent to: writeContent(destination, c);
   //writeContent(destination, readContent(source)) }
```

41.2 Review Exercises

Complete the following exercises.

1) Suppose there is a file named “f_data41.2-1.txt” that contains 10 2-digit integers (separated by a single space character). An example of the structure of the file is shown here.

   ```
   13 55 12 61 12 19 80 91 15 16
   ```

   Write a Java program that calculates and displays the average value of those that are greater than 50.

2) Suppose there is a file named “f_data41.2-2.txt” that contains 3-digit integers (separated by a comma character). An example of the structure of the file is shown here.

   ```
   130,501,322,415,527,191
   ```

   Write a Java program that calculates and displays the average value of those that are between 300 and 500. Assume there is at least one number in the file.

3) Suppose there is a file named “f_data41.2-3.txt” that contains the grades and the full names of the students of a class (separated by a comma character), one record on each line. An example of the structure of the file is shown here.

   ```
   96,George Papas
   100,Anna Maria Garcia
   89,Peter Smith
   .
   .
   ```

   Write a Java program that finds and displays the full name of the best and the worst student of the class. Assume there is at least one record in the file and that all of the grades are different.

4) The IT administrator of a transportation company needs a program to extract useful information from a file named “f_data41.2-4.txt” regarding the items the company transports. Suppose the file contains the width, length, height and description of each item. The dimensions of the items are in inches and each dimension occupies 4 characters in the file (2 characters for the integer part, one for the decimal point and one for the decimal digit). An example of the file's structure is shown below:
110.5 011.2 020.9 Box No 37 (Plastic bottles) 022.6 023.1 040.2 Container No 23 (6 glasses) 009.5 156.6 020.0 Package No 12 (Fragile items) 024.2 020.1 030.1 Container No 29 (Glass bottles)

Write a Java program that: i) prompts the user to enter a keyword to search within the description of the items. For example, if the user enters the word “glass”, then the program must display the following messages `Keyword 'glass' found!`

```
Container No 23 (6 glasses) – Dimensions: 22.6 x 23.1 x 40.2

Container No 29 (Glass bottles) - Dimensions: 24.2 x 20.1 x 30.1
```

ii) finds and displays the volume (in cubic feet) of each item. The messages must be formatted as in the example below: `Volume of each item: Box No 37 (Plastic bottles): Volume = 14.9686 cubic feet Container No 23 (6 glasses): Volume = 12.1451 cubic feet Package No 12 (Fragile items): Volume = 17.2187 cubic feet Container No 29 (Glass bottles): Volume = 8.472940 cubic feet`

iii) finds and displays the total volume (in cubic feet) of all the items.

iv) finds and displays the description of the box with the greatest volume.

Keep in mind that one cubic foot is equivalent to 1728 cubic inches.

5) Write a Java program that prompts the user to enter the filenames of two files. The program must then concatenate the contents of the two files and write the concatenated text in a third file named “final.txt”, placing the contents of the first file after the contents of the second

file. If the user-provided filenames do not contain the ".txt" extension the program must display an error message.

6) Suppose there is a file named "f_data41.2-6.txt" that contains 15 numbers, one on each line. Write a Java program that sorts those numbers in ascending order using the bubble sort algorithm and writes the sorted values in the same file, below the initial unsorted values.

7) Suppose there is a file named "f_data41.2-7.txt" that contains names of eight cities as well as their maximum temperatures on a specific day. An example of the structure of the file is shown here.

```
New York
82.3
Washington DC
84.3
.
.
```

Thus, the odd-numbered lines contain city names and the even-numbered lines contain the maximum temperature of each city. Write a Java program that reads the file line by line and stores the city names and the temperatures in the arrays `cities` and `temperatures` correspondingly. Assume there is at least one name of a city and its corresponding temperature in the file. The program must then

i) calculate and display the average temperature of all cities.

ii) find and display the highest temperature as well as all city names that have this temperature.

8) Some words such as "revolutionary" and "internationalization" are so lengthy that writing them out repeatedly can become quite tiresome. Let's consider a word too long if its length is more than 10 characters. In such cases, this word must be replaced with a special abbreviation which is made like this: you keep the first and the last letter of the word and insert the number of letters between them. For instance, "revolutionary" becomes "r11y" and "internationalization" becomes "i18n".

Suppose there is a file named "f_data41.2-8.txt" that contains an English text. Do the following: i) Write a subprogram named `abbreviate` that accepts a word and when it is more than 10

characters long, it returns its abbreviation; it must return the same word otherwise.

ii) Using the subprogram cited above, write a Java program that reads the text from the file and displays it with all long words replaced by their abbreviations.

Assume that the words are separated by a single space character.

9) Pig Latin is a playful language game often used in English-speaking countries. It involves altering the letters of a word based on a set of simple rules. Here are the rules for translating a word into Pig Latin:

► If the word begins with a vowel, simply add “way” to the end of the word. For example, “apple” becomes “appleway”.

► If the word begins with one or more consonants, move the consonant(s) to the end of the word and add “ay”. For example, “banana” becomes “ananabay” and “flower” becomes “owerflay”.

Suppose there is a file named “f_data41.2-9.txt” that contains an English text. Do the following: i) Write a subprogram named `pigLatinTranslator` that accepts an English word and returns the corresponding Pig Latin translation.

ii) Using the subprogram cited above, write a Java program that reads the text from the file and writes the corresponding Pig Latin translation in a file named “pig_latin_translation.txt”.

Assume that the text contains only lowercase characters of the English alphabet and the words are separated by a single space character.

10) Given two strings, `X = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"` and `Y = "JKWCTAMEDXSLFBYUNG RZOIQVHP"`, you can encrypt any message. The person who holds the `Y` string can decrypt the encrypted message by mapping the letters of string `X` to string `Y`, one by one. To be more specific, the letter “A” is encrypted as “J”, the letter “B” as “K”, the letter “C” as “W”, and so on. According to this encryption method, write a Java program that prompts the user to enter a message and then writes the encrypted message in a file named “encrypted.txt”. Space characters must not be encrypted and should remain as is in the encrypted message.

11) Write a Java program that decrypts the message of the previous exercise (in the file “encrypted.txt”) and writes the decrypted message in a file named “decrypted.txt”.

12) Write a subprogram named `copyFile` that accepts two arguments (`source` and `destination`) and then creates a new copy of the file `source` using the name `destination`.

13) Suppose there is a file named “f_data41.2-13.txt” that contains the lengths of all three sides of a triangle (one on each line). An example of the structure of the file is shown here.

```
16.0
20.6
22.7
```

Do the following i) Write a class named `Triangle` that includes

a) three private float (real) fields named `_sideA`, `_sideB`, and `_sideC`.

b) a constructor that reads the three sides from the file and assigns them to the fields `_sideA`, `_sideB`, and `_sideC`.

c) a public Boolean method named `canBeTriangle` that checks and returns `true` when the values in fields `_sideA`, `_sideB`, and `_sideC` can be lengths of the three sides of a triangle. It must return `false` otherwise.

Hint: In any triangle, the length of each side is less than the sum of the lengths of the other two sides.

d) a public void method named `displayLengths` that displays the lengths of all three sides as well as a message indicating whether those lengths can be lengths of the three sides of a triangle or not.

e) a public void method named `displayArea` that, in case the lengths can be lengths of the three sides of a triangle, calculates and displays the area of the triangle. You can use Heron's formula, which has been known for nearly 2,000 years!

$$Area = \sqrt{S(S-A)(S-B)(S-C)}$$

where *S* is the semi-perimeter $S = \frac{A+B+C}{2}$

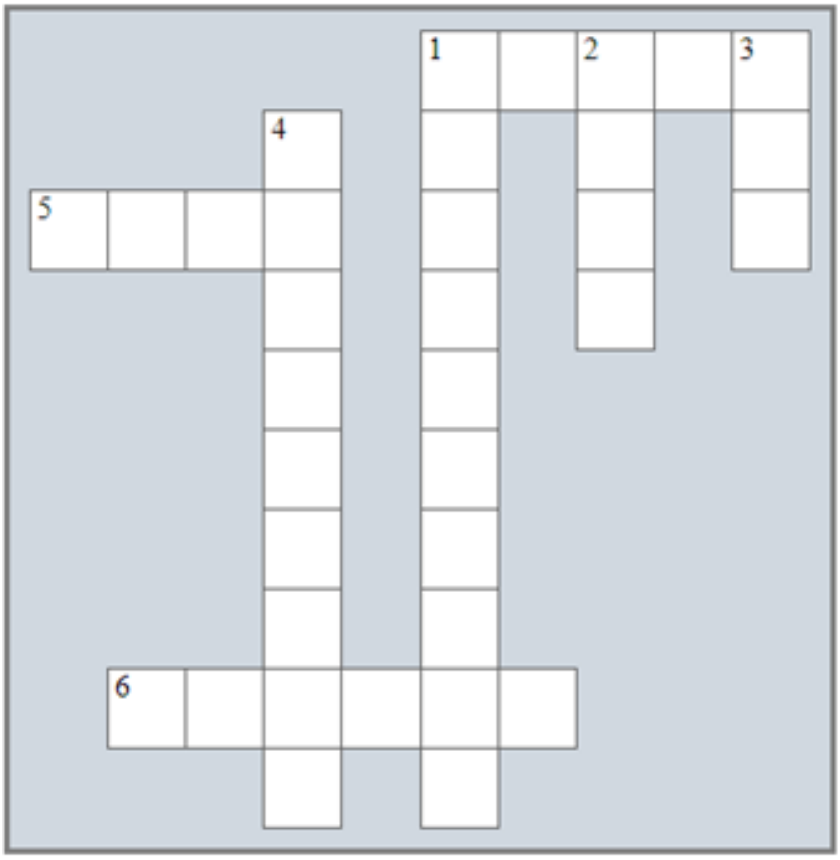
f) a public void method named `displayPerimeter` that, in case the lengths can be lengths of the three sides of a triangle, calculates and displays the perimeter of the triangle.

ii) Using the class cited above, write a Java program that displays all available information about the triangle.

Review in “Files”

Review Crossword Puzzle

1) Solve the following crossword puzzle.



Across

1) After completing reading from a file, you always need to _____ the file.

5) This pointer is quite similar to the index of an array.

6) There are two kinds of files, text files and _____ files.

Down

1) This is what a text file contains.

2) This operation must be performed before any reading/writing operation.

3) When you open a file for appending, the file pointer is positioned at the ____ of the file.

4) A file is stored in this kind of storage device.

Review Questions

Answer the following questions.

1) What is a text file?
2) What is a text file useful for?
3) What exactly does the method `close()` do?
4) What is the difference between `nextLine()` and `nextInt()` methods?
5) Describe the process of iterating through the contents of a file and give an example.

Some Final Words from the Author

I hope you thoroughly enjoyed reading this book. I made every possible effort to ensure it is beneficial and comprehensible, even for people who may have no prior experience in programming.

If you found this book valuable, please consider visiting the web store where you purchased it, as well as goodreads.com, to show your appreciation by writing a positive review and awarding as many stars as you think appropriate. By doing so, you will motivate me to keep writing and, of course, you'll be assisting other readers in discovering my work.

And always remember: Learning is a lifelong, continuous process that begins at birth and extends throughout your lifetime!

Footnotes

[1] The word "algorithm" derives from the word "algorism" and the Greek word "arithmos". The word "algorism" comes from the Latinization of the name of Al–Khwārizmī[2] whereas the Greek word “arithmos” means “number”.

[2] Muḥammad ibn Al–Khwārizmī (780-850) was a Persian mathematician, astronomer, and geographer. He is considered one of the fathers of algebra.
[RETURN]

[3] Donald Ervin Knuth, (1938-), is a prominent American computer scientist and mathematician, renowned as the “father of the analysis of algorithms”. He authored the influential multi-volume work, The Art of Computer Programming, and made groundbreaking contributions to computational complexity analysis and literate programming.
[RETURN]

[4] Corrado Böhm (1923-2017) was a computer scientist known especially for his contribution to the theory of structured programming, and for the implementation of functional programming languages.
[RETURN]

[5] Giuseppe Jacopini (1936-2001) was a computer scientist. His most influential contribution is the theorem about structured programming, published along with Corrado Böhm in 1966, under the title *Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules*.
[RETURN]

[6] Grace Murray Hopper (1906-1992) was an American computer scientist and US Navy admiral. She was one of the first programmers of the Harvard Mark I computer, and developed the first compiler for a computer programming language known as A–0 and later a second one, known as B–0 or FLOW-MATIC.
[RETURN]

[7] George Boole (1815-1864) was an English mathematician, philosopher, and logician. He is best known as the architect of what is now called Boolean logic (Boolean algebra), the basis of the modern digital computer.
[RETURN]

[8] Daniel Gabriel Fahrenheit (1686-1736) was a German physicist, engineer, and glass blower who is best known for inventing both the alcohol and the mercury thermometers, and for developing the temperature scale now named after him.
[RETURN]

[9] William Thomson, 1st Baron Kelvin (1824-1907), was an Irish-born British mathematical physicist and engineer. He is widely known for developing the basis of absolute zero (the Kelvin temperature scale), and for this reason a unit of temperature measure is named after him. He discovered the Thomson effect in thermoelectricity and helped develop the second law of thermodynamics.
[RETURN]

[10] Anders Celsius (1701-1744) was a Swedish astronomer, physicist, and mathematician. He founded the Uppsala Astronomical Observatory in Sweden and proposed the Celsius temperature scale, which takes his name.
[RETURN]

[11] Heron of Alexandria (c. 10-c. 70 AD) was an ancient Greek mathematician, physicist, astronomer, and engineer. He is considered the greatest experimenter of ancient times. He described the first recorded steam turbine engine, called an “aeolipile” (sometimes called a "Hero engine"). Heron also described a method of iteratively calculating the square root of a positive number. Today, though, he is known best for the proof of “Heron's Formula” which finds the area of a triangle from its side lengths.
[RETURN]

[12] Pythagoras of Samos (c. 571-c. 497 BC) was a famous Greek mathematician, philosopher, and astronomer. He is best known for the proof

of the important Pythagorean theorem. He was an influence for Plato. His theories are still used in mathematics today.
[RETURN]

[13] William Shakespeare (1564-1616) was an English poet, playwright, and actor. He is often referred to as England's national poet. He wrote about 40 plays and several long narrative poems. His works are counted among the best representations of world literature. His plays have been translated into every major living language and are still performed today.
[RETURN]

[14] A quantity that is either zero or positive.
[RETURN]

[15] Francis Beaufort (1774-1857) was an Irish hydrographer and officer in Britain's Royal Navy. He is the inventor of the Beaufort wind force scale.
[RETURN]

[16] A quantity that is either zero or negative.
[RETURN]

[17] The value of −459.67° (on the Fahrenheit scale) is the lowest temperature possible and it is called *absolute zero*. Absolute zero corresponds to −273.15 °C on the Celsius temperature scale and to 0 K on the Kelvin temperature scale.
[RETURN]

[18] A quantity that is either zero or negative.
[RETURN]

[19] Madhava of Sangamagrama (c. 1340-c. 1425), was an Indian mathematician and astronomer from the town of Sangamagrama (present day Irinjalakuda) of India. He founded the Kerala School of Astronomy and Mathematics and was the first to use infinite series approximations for various trigonometric functions. He is often referred to as the “father of

mathematical analysis".
[RETURN]

[20] Gottfried Wilhelm von Leibniz (1646-1716) was a German mathematician and philosopher. He made important contributions to the fields of metaphysics, logic, and philosophy, as well as mathematics, physics, and history. In one of his works, *On the Art of Combination (Dissertatio de Arte Combinatoria*), published in 1666, he formulated a model that is considered the theoretical ancestor of modern computers.
[RETURN]

[21] Leonardo Pisano Bigollo (c. 1170-c. 1250), also known as Fibonacci, was an Italian mathematician. In his book *Liber Abaci* (published in 1202), Fibonacci used a special sequence of numbers to try to determine the growth of a rabbit population. Today, that sequence of numbers is known as the Fibonacci sequence. He was also one of the first people to introduce the Arabic numeral system to Europe; this is the numeral system we use today, based on ten digits with a decimal point and a symbol for zero. Before then, the Roman numeral system was being used, making numerical calculations difficult.
[RETURN]

[22] Brook Taylor (1685-1731) was an English mathematician who is best known for the Taylor series and his contributions to the theory of finite differences.
[RETURN]

[23] Samuel Finley Breese Morse (1791-1872) was an American painter and inventor. Morse contributed to the invention of a single-wire telegraph system and he was a co-developer of the Morse code.
[RETURN]

[24] In Greek mythology, the Titans and Titanesses were the children of Uranus and Gaea. They were giant gods who ruled during the legendary Golden Age (immediately preceding the Olympian gods). The male Titans were Coeus, Oceanus, Crius, Cronus, Hyperion, and Iapetus whereas the

female Titanesses were Tethys, Mnemosyne, Themis, Theia, Rhea, and Phoebe. In a battle, known as the Titanomachy, fought to decide which generation of gods would rule the Universe, the Olympians won over the Titans!
[RETURN]

More...

This is the nested decision control structure
[RETURN]

This is a nested case decision structure
[RETURN]

This is a nested single-alternative decision structure
[RETURN]

This is a nested dual-alternative decision structure
[RETURN]

This statement is not affected by the previous decision control structure and does not affect the next one.
[RETURN]

The previous and next decision control structures are affected by this statement
[RETURN]

Code Fragment 1
[RETURN]

Code Fragment 1
[RETURN]

The destination is inside the country. Check the weight and calculate the corresponding shipping cost.
[RETURN]

The destination is outside the country. Check the weight and calculate the corresponding shipping cost.

[RETURN]

$-5 < x \leq 0$
[RETURN]

$0 < x \leq 6$
[RETURN]

$6 < x \leq 20$
[RETURN]

All other values of x
[RETURN]

Code Fragment 1
[RETURN]

This pair of statements is executed 4 times forcing the user to enter 4 numbers.
[RETURN]

This is the part of the program that somehow repeats.
[RETURN]

This must be written 20 times
[RETURN]

Nested loop
[RETURN]

This is the dual-alternative decision structure
[RETURN]

This is the post-test loop structure
[RETURN]

This is the dual-alternative decision structure
[RETURN]

A statement or block of statements 1
[RETURN]

A statement or block of statements 2
[RETURN]

A statement or block of statements 1
[RETURN]

A statement or block of statements 2
[RETURN]

A statement or block of statements 1
[RETURN]

This code fragment calculates the denominator.
[RETURN]

Code Fragment 1
[RETURN]

Data input stage without validation.
[RETURN]

Data input validation without error messages.
[RETURN]

Data input validation with one single error message.
[RETURN]

Data input validation with a different error message for each type of input error.
[RETURN]

Data input stage without validation
[RETURN]

Data input validation with one single error message
[RETURN]

Data input validation
[RETURN]

Code Fragment 1
[RETURN]

Data input stage without validation.
[RETURN]

Data input and validation
[RETURN]

Data input and validation
[RETURN]

Data input and validation
[RETURN]

This is a formal argument list
[RETURN]

This is an actual argument list
[RETURN]

This is an actual argument list
[RETURN]

By default, arrays in Java are passed by reference.
[RETURN]